Matching of Ontologies with XML Schemas using a Generic Metamodel

Christoph Quix, David Kensche, Xiang Li

RWTH Aachen University, Informatik 5 (Information Systems), 52056 Aachen, Germany
{quix,kensche,lixiang}@cs.rwth-aachen.de

Abstract. Schema matching is the task of automatically computing correspondences between schema elements. A multitude of schema matching approaches exists for various scenarios using syntactic, semantic, or instance information. The schema matching problem is aggravated by the fact that models to be matched are often represented in different modeling languages, e.g. OWL, XML Schema, or SQL DDL. Consequently, besides being able to match models in the same metamodel, a schema matching tool must be able to compute reasonable results when matching models in heterogeneous modeling languages. Therefore, we developed a matching component as a part of our model management system *GeRoMeSuite* which is based on our generic metamodel *GeRoMe*. As *GeRoMe* provides a unified representation of models, the matcher is able to match models represented in different languages with each other. In this paper, we will show in particular the results for matching XML Schemas with OWL ontologies as it is often required for the semantic annotation of existing XML data sources.
GeRoMeSuite allows for flexible configuration of the matching system; various matching algorithms for element and structure level matching are provided and can be combined freely using different ways of aggregation and filtering in order to define new matching strategies. This makes the matcher highly configurable and extensible. We evaluated our system with several pairs of XML Schemas and OWL ontologies and compared the performance with results from other systems. The results are considerably better which shows that a matching system based on a generic metamodel is favorable for heterogeneous matching tasks.

1 Introduction

Integration of information systems is a major challenge that has been addressed in several disciplines such as database and semantic web research. One of the key issues in integration is creating a mapping between the data models of the systems involved. This work is, for example, required if the data from different data sources must be merged in a data warehouse or if two e-business systems must communicate with each other.

Schema matching is the task of identifying a set of correspondences (also called a morphism or a mapping) between schema elements. Many aspects have to be considered during the process of matching, such as data values, element names, constraint information, structure information, domain knowledge, cardinality relationships, and so on. All this information is useful in understanding the semantics of a schema, but it can be a very time consuming problem to collect this information. Therefore, automatic methods are required for schema matching.

A multitude of methods have been proposed for schema matching [24,26] using different types of information to identify elements or focusing on models represented in a specific modeling language such as the Relational Data Model, XML Schema, or OWL [6,8,9,17,18]. To avoid confusion with the terms being used in this paper (e.g. metamodel, model, schema), we want to clarify the terminology first. We will use the terminology defined in the IRDS standard [10] and used by the Object Management Group [22]. According to the IRDS standard, models, schemas, and ontologies are all on the same level and describe the structure of data instances. A metamodel (or a modeling language) is used to define a model, schema or ontology. Examples for metamodels are OWL, UML, or the Relational Data Model.

The schema matching problem is aggravated by the fact that models employed by one system are often represented in different modeling languages. Consequently, besides being able to match models in the same metamodel, a schema matching tool must be able to compute reasonable results when matching models in heterogeneous modeling languages. This is for example required for the annotation of existing XML or relational data sources with ontologies, to enable semantic queries to these sources. Another example is the enrichment of XML web services with semantic information to get semantic web services.

In this paper, we present the matching system which is part of our generic model management system *GeRoMeSuite* [14,15,23]. *GeRoMeSuite* is based on our role-based metamodel *GeRoMe* ([13], phonetic transcription: dʒerəʊm) which provides a generic but yet detailed representation of models represented in different modeling languages. By using *GeRoMe*, our system is able to match models expressed in heterogeneous modeling languages which we will apply in this paper to the case of matching XML schemas with OWL ontologies.

Currently, schema matching systems represent models as directed labeled graphs to support the matching of models from different metamodels. However, the way how a model is encoded as graph is crucial for the match result as structural similarities are also important in schema matching. As models from different metamodels are represented differently in graphs (different labels, different structures), the matching between such models produces often poor results. As we will show, *GeRoMeSuite* produces significantly better results for matching models from heterogeneous metamodels which indicates an advantage of using a generic metamodel for the representation of models. We evaluated the matching performance of our system using various examples for matching OWL ontologies with XML schemas.

The main contributions of our work are (i) a system for matching models using a true generic representation, (ii) which provides several matchers and traversal strategies, and (iii) is based on a very flexible and easily extensible implementation. The generic representation of schemas allows us to apply our implementations of matching algorithms to any combination of models regardless of the modeling languages that the models are originally represented in. Furthermore, the high level of detail of our generic representation enables us to provide different structural views on a model to structure-level matching algorithms. In doing so structural matchers can, for instance, incorporate into their similarity assessment associations between types or derivations between types (the IsA-hierarchy) or even both.

The structure of the paper is as follows. In the next section, we will discuss existing approaches to schema matching. Then, we will describe briefly our generic metamodel *GeRoMe* using an ontology and an XML schema as example. Section 4 presents the system we have developed in terms of its architecture and implemented matchers. In section 5, we present the evaluation of our system. We also discuss and analyze the results of the tested schema matching systems. We conclude our paper with a discussion of our approach and an outlook to future work.

2 Related Work

There have been many approaches to schema matching. The main reason for the various approaches is that each matching problem has its own characteristics and might require a specific solution. The approaches to schema matching can be distinguished by the information they use: some focus only on the schemas, some use external information in form of thesauri, dictionaries or acronym databases, and, if available, it is also possible to use the instance data to find similarities between schemas [24,26].

The Cupid algorithm [18] is intended to be generic across data models and has been applied to XML and relational examples. It uses auxiliary information sources for synonyms, abbreviations, and acronyms. It implements a generic schema matching algorithm combining linguistic and structural schema matching techniques. The input schemas are encoded as graphs. Nodes represent schema elements and are traversed in a combined bottom-up and top-down manner. The matching algorithm consists of three phases. The first phase (linguistic matching) computes linguistic similarity coefficients between schema element names (labels). The second phase (structural matching) computes structural similarity coefficients which measure the similarity between contexts in which individual schema elements occur. The main idea behind the structural matching algorithm is to propagate the similarity of leaf items to the similarity of inner nodes. Finally, the third phase (mapping generation) computes weighted similarity coefficients and generates final mappings by choosing pairs of schema elements with weighted similarity coefficients which are higher than a threshold.

A similar idea is followed by the Similarity Flooding algorithm [20]. Schemas are also represented as directed labeled graphs. Based on the idea that if two nodes are similar then also their neighbors are similar, the similarity of two nodes in the graph is propagated to its neighbors. This procedure is repeated until the Euclidean distance between two subsequent similarity matrices is below a certain threshold. The initial input similarities can be computed by any kind of (linguistic) matching method. The algorithm can be applied to arbitrary graph structures. In [20], there are also several strategies proposed to filter the mapping pairs from the computed similarity values.

The COMA schema matching system is a platform designed to combine multiple matchers in a flexible way [6]. It provides a large number of individual matchers, which contains both terminology approaches and structural approaches. After combining the mapping results from the individual matchers, the output mapping could be chosen as the final result or reused as an individual matching result. As a generic matching system, COMA accepts different schema types as input, such as XML schemas and relational schemas, which are internally represented as directed graphs. COMA also allows users

to reuse the previously obtained matching results. COMA++ [2] is an extended and improved update of the COMA system. It supports ontologies as inputs and provides several matchers for ontology matching.

Compared to other ontology alignment tools, COMA++ produces also very good results in the area of ontology alignment [19]. In principle, ontology matching can use the same ideas as schema matching (e.g. a combination of linguistic and structural matchers). However, as ontologies contain usually more semantic information and constraints than schemas, methods for ontology matching can also use this information to detect similarities between ontologies [12,21]. For example, in [7] a metric for the similarity of concepts is defined using properties, restrictions, sub- and super-class relationships, and so on. There are several tools for ontology alignment, which were also evaluated in the Ontology Alignment Evaluation Initiative 2006 (`http://oaei.ontologymatching.org/2006/`). Some of them performed better than COMA++ (e.g. Falcon-AO [9] and RiMOM [17]), but these tools are not able to match XML schemas with ontologies.

The ARTEMIS tool [4] for schema integration comes closest to our approach. It also uses a generic metamodel (called reference data model in their work) which is the relational metamodel with some additional object oriented features. Using this generic metamodel, they can uniformly analyze models represented as relational, EER or object oriented models. The matching component matches elements based on their name, data type or structural similarity. To deal with synonyms and hypernyms, the tool uses also an external thesaurus (WordNet). However, ARTEMIS has not been applied to match OWL ontologies and XML schemas. As the tool is not available anymore on the Internet, we could not compare it with our matching system. To the best of our knowledge, COMA++ is the only tool, which is available for download and allows to match XML schemas with OWL ontologies.

3 The Generic Metamodel *GeRoMe*

The *Ge*neric *Ro*le based *Me*tamodel *GeRoMe* [13] uses *role based* modeling. Each model element of a native model (e.g. an XML Schema or a relational schema) is represented as an object that plays a set of roles which decorate it with features and act as interfaces to the model element. We wiil briefly introduce the main ideas of *GeRoMe* by using an example representing an XML schema, which we will use also later as a running example. Representing XML Schema in a generic metamodel is quite challenging as it supports modeling constructs which are not common in other metamodels, such as ordered elements or the derivation of simple types by regular expressions. The example schema contains three complex types (AirlineType, EmpType, and PilotType) and three elements (Airline, Employee, and Pilot). Employees work for an airline, pilots are modeled as a subtype of employee and have an additional attribute Lic_Num.

The *GeRoMe* representation in fig. 1 shows each model element as a *ModelElement* object (gray rectangle) which plays a number of roles (white squares). Each such role object may be connected to other roles or literals, respectively. For the sake of readability, we refrain here from showing the whole model and omitted repeating structures with the same semantics such as *Visible* roles.

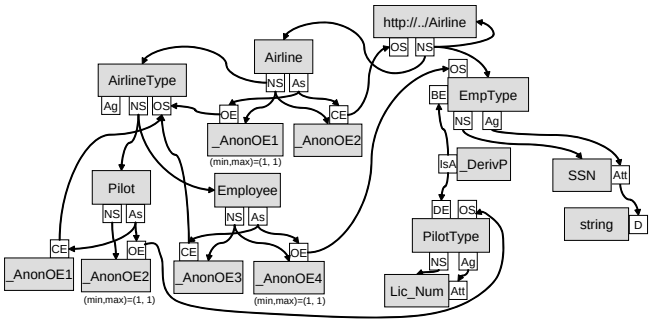

Fig. 1. *GeRoMe* representation of an XML schema

The XML Schema element is an association between its enclosing type and the complex type of the nested element. It is always a 1:n association since an XML document is always tree structured. Because of this, the elements in XML Schema are represented by associations in *GeRoMe*. In the example, the elements Airline, Employee and Pilot play *Association* (As) roles connecting the model elements corresponding to the complex types AirlineType, EmpType, and PilotType via anonymous *ObjectAssociationEnd* (OE) and *CompositionEnd* (CE) roles. The *CompositionEnd* role refers to the enclosing complex type of the XML element. The root element Airline is a special case; as it is not enclosed in a complex type, the *CompositionEnd* role for the enclosing type points to the model element http://../Airport representing the schema.

Model elements defined within other model elements such as attributes and XML elements are referenced by the *Namespace* (NS) role of the containing element. For example, the element Employee is owned by the Namespace role of *AirlineType*. Furthermore, the complex types play *Aggregate* (Ag) roles, as they can have attributes, and *ObjectSet* (OS) roles, as they can participate in associations. For example, the *Attribute* (Att) roles of SSN and Lic_Num are connected to the *Aggregate* role of the corresponding model element. Finally, the subtype relationship between PilotType and EmpType is represented by a separate anonymous model element _DerivP. This model element plays an *IsA* role which is connected to the *BaseElement* (BE) role of EmpType and to the *DerivedElement* (DE) role of PilotType.

We have to admit that the *GeRoMe* representation of a model is not easy to understand, but this representation is used only internally in a model management tool; the user will still use her favorite modeling language. The complexity of *GeRoMe* is caused by the complexity of the original modeling languages which can be represented in *GeRoMe*. To be able to represent the details of several modeling languages in a generic way, these details have to be present in *GeRoMe* as well.

The benefit of this generic and detailed representation is that modeling constructs from different metamodels which have equivalent (or similar) semantics are represented by the same roles in *GeRoMe*. This means that the structure of *GeRoMe* models, even if they are originally represented in different metamodels, is similar if they model the same domain. This structural similarity is very important for schema matching as we will show in the evaluation of our matching system in section 5.

4 Schema Matching in *GeRoMeSuite*

Based on the generic metamodel *GeRoMe*, we implemented a schema matching system with the aim to have an extensible and flexible framework for matching models regardless of the modeling language they are represented in. Our system can use any model that can be imported into the generic modeling language as input for the match operation. Currently, this includes relational models, XML Schemas and OWL ontologies.

Another requirement was that the system is built up from components that can be easily combined to new composite algorithms. *GeRoMeSuite* contains a set of *graph traversal* strategies that provide different views on the same model. For each provided graph traversal, there is a corresponding *tree traversal* that can be used if a tree structure is needed. These different views on the structure of the same model influence the results of structure level matchers. Besides allowing variations of the data structure being matched, our matchers also consist of arbitrarily combinable and parametrized steps. In the following, we explain some traversal strategies, the central components of a matcher in *GeRoMeSuite* and how to combine these components to a matcher configuration.

4.1 Graph Traversal Strategies

During the development of the matching system, our aim was to exploit the special characteristics of models represented in *GeRoMe*. As shown in the example in section 3, a *GeRoMe* model is a highly connected structure, i.e. model elements are linked by many different types of relationships. These different types of relationships can be used to define the structure used by the structure level matchers. For example, the links between *Association*, *Aggregate*, and *Attribute* roles could be used to define such a structure. However, there are also other possibilities: the structure implied by the *Namespace* roles define the context in which model elements are defined; *IsA* and other derivation roles can be used to build up a type hierarchy.

In order to use these different structures in our matcher, we defined different iterators for *GeRoMe* models which implement certain traversal strategies, i.e. they navigate a model in a specific way. Our current implementation provides five traversal strategies:

Namespace: Uses the *Namespace* roles to navigate the model.
Derivation: Builds a type (or class) hiearchy.
Association: Uses mainly *Association* roles to navigate the model (e.g. XML Schemas are represented as trees as in most XML editors).
Types: Like the *Association* iterator, but includes also the model elements representing types (e.g. *Aggregates*, *ObjectSets*, *Domains*).
Structure: Reproduces the complete structure of a *GeRoMe* model.

These iterators usually produce graph structures; for matchers which require tree structures as input, we implemented a "meta"-iterator which produces a tree structure from a graph. In addition, we can restrict the iterators to return only model elements which play a *Visible* role (i.e. elements which have an explicit name). Thus, in total we provide twenty different iteration strategies.

Fig. 2 shows the *Types* and *Namespace* traversals for the model from fig. 1. As it can be seen from the example, the traversals imply a different structure as the semantics

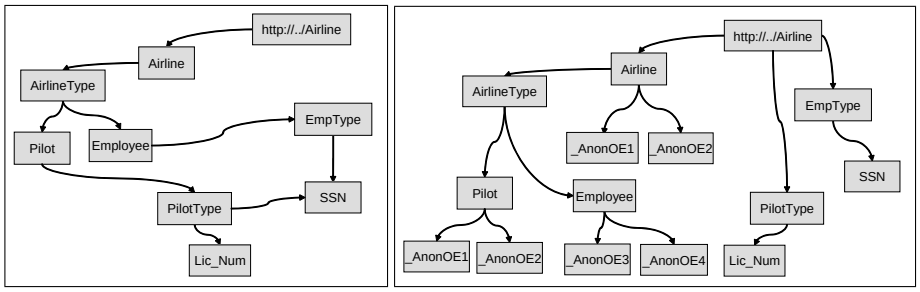

Fig. 2. Types and Namespace traversal for the XML schema from fig. 1

of the relationships considered in a traversal strategy is quite different. The namespace structure is sometimes an "artificial" structure as it does not represent the structure of the data; it is just the structure in which the schema is defined. The right part of fig. 2 shows the namespace iterator, in which EmpType and PilotType are directly connected to the root of the schema, although the data of these would be nested under an Airline element. A traversal of the model corresponding to the structure of the instance data is produced by the *Types* traversal strategy which is shown in the left part of fig. 2.

4.2 Matcher Components

Schema-based matchers can be classified into element-level and structure-level matchers [24]. Element-level matchers consider an element in isolation, whereas structure-level matchers also consider the context of an element. Additionally to these two kinds of matcher components, *GeRoMeSuite* provides different strategies for aggregation of multiple input morphisms to one output morphism and different filters for morphisms.

Element Level Matchers are capable of computing an initial morphism between two models from scratch. They get two models as input and return a morphism between these two models. Usually, they are based on assessment of similarity for pairs of single model elements. Most such matchers perform a string comparison on the names of the elements using some metric. Whereas this assessment depends in most cases on the two elements alone, it is possible to incorporate the model structure into this step as well. For instance, the similarity of two elements may be determined by the similarity of the possible paths to this element through the model.

GeRoMeSuite provides two basic element level matchers. The `StringMatcher` compares two model elements without taking into account their structure. It is parameterized with a metric that gets two model elements for input and returns a similarity value. Currently, we provide the Levenshtein metric [16] (or edit distance), the Jaro/Winkler metric [11,28], and an improved string matcher [27]. In the future we also plan to add a datatype matcher that can assess the similarity of primitive datatypes.

On the other hand, the `NamepathMatcher` also takes into account the structure of the two models to be matched. It applies the aforementioned string similarity metrics to a set of path expressions that lead to a model element. As its similarity assessment

is based on paths to the respective model elements it also requires a tree traversal as a parameter. To assess similarity of two model elements, the `NamepathMatcher` computes the similarity of each pair of paths to the elements and then combines (based on a configurable strategy) these values to the final similarity assessment.

Structure Level Matchers refine an input morphism based on some strategy and on the structures of the models to be matched. That is, they receive a single morphism as input and return a single morphism as output. The general idea of structure level matchers is that the similarity of neighboring elements contributes to the similarity of the element itself. This idea is, for example, realized in the Similarity Flooding algorithm [20] which is also implemented in our system. In addition, our schema matcher provides a *children* matcher. The children matcher resembles the idea of the Cupid algorithm [18]; if the children of an element A are similar to the children of an element B, then A and B are also similar. Both structure level matchers require a graph traversal as input. Furthermore, they are composed of a variety of exchangeable strategy objects that implement certain parts of the respective matching algorithms.

Our schema matching system relies on well-known schema matching methods. The goal of this work is not to provide new algorithms for schema matching, but the usage of a generic metamodel for schema matching and the proof that the generic representation of models is beneficial.

Aggregation Strategies can be used to combine an arbitrary number of morphisms to a single morphism using average, maximum or weighted similarities of model elements.

Morphism filters select similarity values from morphisms based on various criteria such as the maximum distance to the best match, keeping only at least the best K matches, or applying a simple threshold to the similarity values. These filters can be adjusted for existing morphisms to mask or unmask links, but they can also be used as an intermediate step in a matcher to refine the input of subsequent steps.

4.3 Matcher Configuration

Fig. 3 shows an example of how to configure a matcher using the aforementioned components. An arbitrary number of matcher components can be chosen from the set of all matchers already defined by the user and the predefined matchers. In the same way filter and aggregation steps are added to the matcher. Each component has a result morphism and one or more input morphisms. Furthermore, each matching component may provide a GUI class that fills a configuration window with its own controls for specification of its parameters. When all required parameters have been defined the matcher configuration is stored in a configuration repository and is then available for execution and as a component of future custom matchers.

Extensibility Our matching subsystem is easily extensible. Predefined components such as the different graph and tree traversals or the metrics can be reused for new

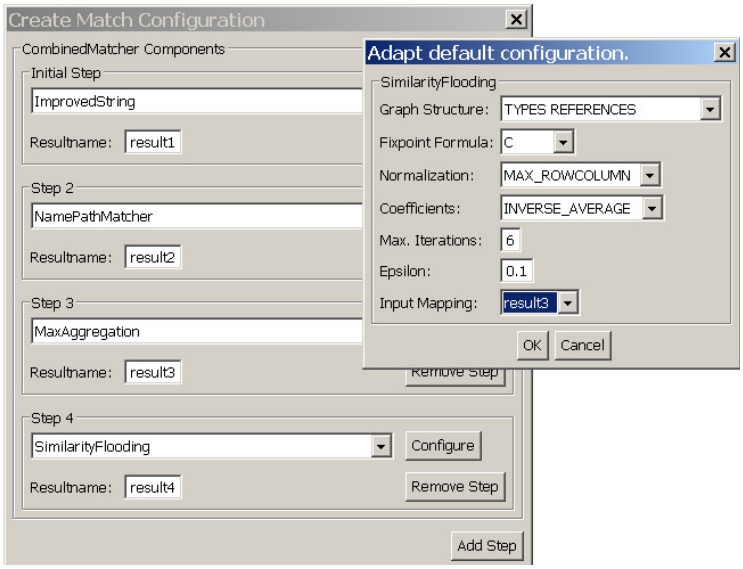

Fig. 3. Creating a matcher configuration

component classes. All interfaces of the available matcher steps are clearly defined and consolidated. For instance, adding a new filter requires only the implementation of the filter functionality (currently the largest is 32 LOC) and the provision of the user interface (currently the largest is 25 LOC).

Adding new matching algorithms is also easily possible, it just requires the creation of a subclass of an abstract *Matcher* class and the implementation of the *match* method. For example, the implementation of Similarity Flooding uses less than 1000 LOC of which the largest part is used for the implementation of the propagation graph.

For all components, the user interface definition only consists of filling a panel with controls and adding event listeners that update parameter values in the configuration object. This panel is then available in various stages of the process. For instance, a filter can be used for filter steps of a matcher and for filtering a currently displayed morphism.

User Interface Fig. 4 displays the view of a morphism as it is shown after executing a matcher or loading an existing mapping. Both models are shown as a tree view. The traversal strategy to be used for the tree view can be chosen from a drop-down box.

The morphism itself is shown as a set of lines between the elements of the two models in the center of the view. As in other matching systems different color shades are used to distinguish different degrees of similarity. Links adjacent to selected model elements are displayed in another color. Furthermore, the link(s) with the maximum similarity originating from the selected element is (are) distinguished. To further improve the usability of the system, the user can mask all links that are not adjacent to the currently selected element.

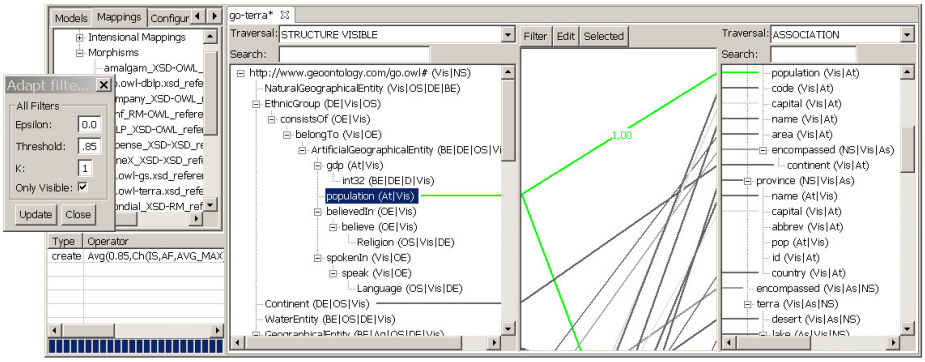

Fig. 4. Viewing and tweaking a morphism

Using a non-modal filtering dialog, all available filters can be adjusted to filter the currently selected morphism. The filters can be freely narrowed and relaxed until a satisfactory result is found before the user starts to manually fine-tune the morphism.

5 Evaluation

The matcher component has been evaluated by gaging the metrics that are usually used for evaluation of schema matching applications [5], that is *precision*, *recall*, *f-measure(0.5)*, and *overall*. The *overall* metric was developed especially for schema matching systems [5]; it should represent the effort to correct the mapping. As adding mappings is more difficult than removing incorrect mappings, it puts more emphasis on recall than on precision.

For the purpose of this paper, we evaluated only examples that involved ontologies and XML schemas. However, we tested our matching system also with several other examples (also involving other modeling languages) which had a similar results in terms of matching performance as the examples shown in here. As COMA++ is the only other system which is able to match XML schemas and ontologies, and is available for us, we could compare our matching system only to COMA++.

The featured tasks are matching `terra.xsd` from the Mondial data set (`http://www.dbis.informatik.uni-goettingen.de/Mondial/`) with a manually created ontology, matching MapOnto's `DBLP.xsd` with a bibtex ontology (`http://cse.unl.edu/~scotth/SWont/bib.owl`), and the company example (`company.xsd` and `company-er.owl`) from the MapOnto project (`http://www.cs.toronto.edu/semanticweb/maponto/`). For all these tasks and configurations tested, our matching system had an execution time of less than 10 seconds.

5.1 Comparison with COMA++

For COMA++ we performed each of these matching tasks with all available combinations of preconfigured matchers and additionally defined new matchers to search for the

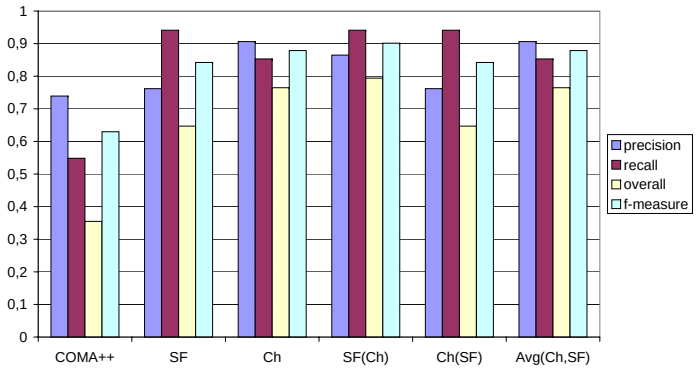

Fig. 5. Comparison of *GeRoMeSuite* with COMA++ for the company example

best possible results. In *GeRoMeSuite* we used basically five different combined matchers. For each of the matchers we used the improved string metric to create an initial match result which was given as input to either the children matcher (Ch) or our similarity flooding implementation (SF). We placed our focus on varying the parameters of these structure level matchers such as traversal strategies or combination of component results to an overall result of the respective matcher. In a next step, we combined these basic matchers in various ways in which we used the best results of the children matcher as input for similarity flooding (SF(Ch)) or vice versa (Ch(SF)) or simply combined the individual result morphisms by computing their average (Avg(Ch, SF)). The following diagrams show the best results of each matcher on the respective match task.

Fig. 5 presents the results of matching the company example, using the metrics precision, recall, overall, and f-measure for COMA++ and the five matchers defined with *GeRoMeSuite*. The company example is a pair of two relatively small models and most elements of the models could be mapped. For COMA++ the best results could be achieved using variations of the original COMA algorithm with different thresholds or other variations of selection strategies. Each of the five matchers of *GeRoMeSuite* outperforms the best result of COMA++ for all quality metrics. Similarity flooding in our implementation achieved better results than the best configuration of COMA++, but was outperformed by the children matcher. However, the best result could be achieved by using the result of the children matcher as initial result for the similarity flooding algorithm (SF(Ch)). The children matcher used the *Association* iteration strategy on this example.

Fig. 6 displays the quality of results for the bibtex/DBLP example. On this example, both tools did not achieve outstanding results. The reason for the poor performance of all matchers is that this matching task is quite difficult as labels and structures of the two models are quite different. *GeRoMeSuite*'s children matcher (Ch) outperformed the best result of COMA++. Whereas its recall is slightly worse, its precision is better by about the same degree. Because the overall metric punishes precision below 0.5, our overall is slightly better. However, the difference is small enough that it seems reasonable to state that both matchers achieve about the same performance. Similarity

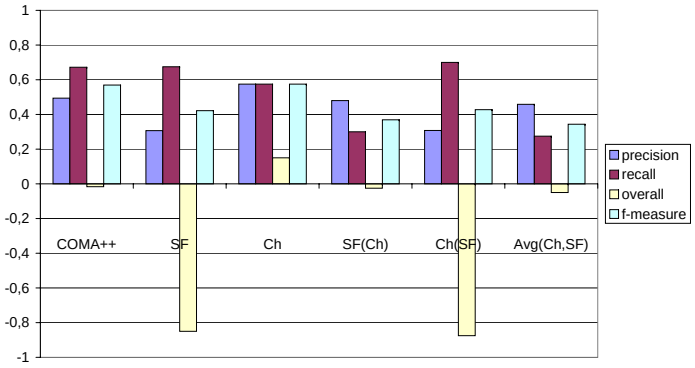

Fig. 6. Comparison of *GeRoMeSuite* with COMA++ for the bibtex example

flooding achieved a very small overall measure due to its low precision on this example. Consequently, the children matcher that receives similarity flooding's results as input (Ch(SF)) performs poor as well. Overall the simple children matcher returned the best result for this example.

The last example is the task of matching the XML Schema `terra.xsd` from the Mondial database with an ontology of the geographical domain. The results are shown in fig. 7. Again, *GeRoMeSuite*'s children matcher by far outperformed the best result of COMA++. Also, similarity flooding was outperformed by the children matcher. However, the averaging of the two results (Avg(Ch,SF)) slightly dominates both input morphisms. The children matcher used the *Association* traversal strategy, similarity flooding used the *Structure* traversal strategy on this example.

Thus, on the given matching tasks *GeRoMeSuite* was at least as good as COMA++ or even outperformed COMA++. However, we must emphasize that we are of course not as familiar with COMA++ as we are with our own matcher component. There is a

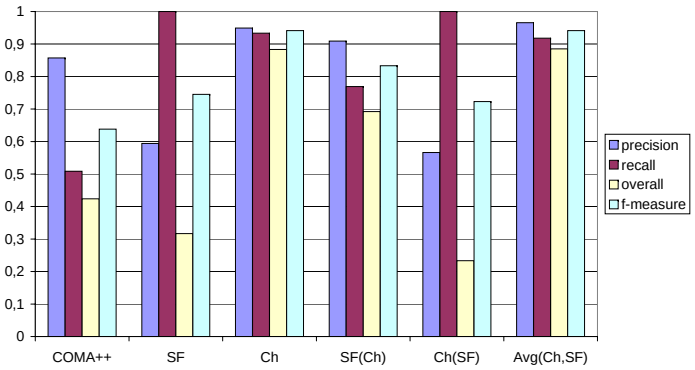

Fig. 7. Comparison of *GeRoMeSuite* with COMA++ for the geographic example

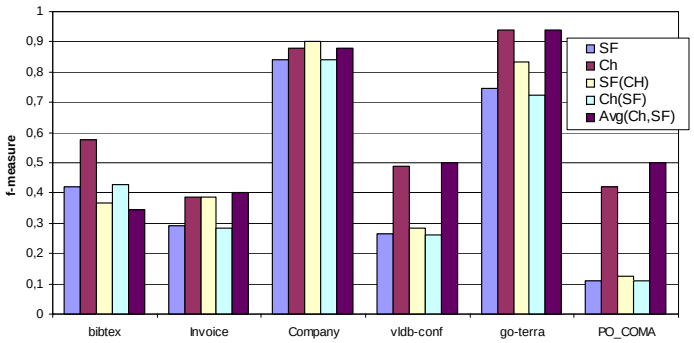

Fig. 8. Quality of matcher results in *GeRoMeSuite* (F-Measure)

large number of configuration options for COMA++ and, consequently, an experienced user may have produced better results with this tool. Nevertheless, we tested more than 50 configurations for COMA++ and presented here only the best results. We tried every configuration using the default matchers and also created some custom matcher configurations searching for more promising results. It is reasonable to assume that comparable results can be achieved for other examples.

In the last example it could be seen that averaging of the two results (Avg(Ch,SF)) dominated both, children matcher and similarity flooding. In fact, our tests on other examples suggest that averaging the results of these two matchers improves the result in many cases.

Fig. 8 compares the results of our matchers in $F\text{-}Measure(0.5)$ for different matching tasks. The combined matchers SF(Ch) and Ch(SF) could not challenge the children matcher alone. However, the simple aggregation by averaging resulted only in one case (bibtex) in a mapping that was inferior to the input mappings, but in all other cases the results had the same or even better quality than the individual matchers alone.

5.2 Effect of Filter Configuration on the Quality

The variation of morphism filters has of course a significant impact on the quality of the result. *GeRoMeSuite* provides four filters for morphisms. The *epsilon* filter allows all links originating at a model element the confidence of which is within a specified range from the element's best match, the *TopK* filter allows only the best k matches for each element, and the *threshold* filter allows links with a confidence measure exceeding a certain value. These filters can be freely configured, whereas the *visible* filter, when enabled, denies all links that involve anonymous model elements such as an anonymous object property that is mapped to a visible property in the other model.

We made the experience that our system is quite stable with respect to variations of the filters, i.e. the results do not vary too much if different filter configurations are applied. Furthermore, the evaluation has shown that if we choose a threshold of about 0.8, the quality of the match result is very close to the best result which can be achieved with our matcher.

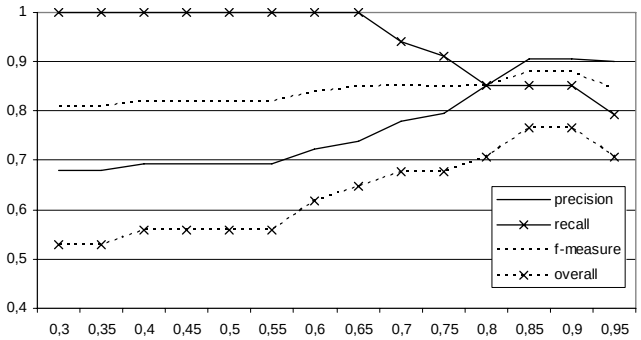

Fig. 9. Matching the company example with different thresholds

For instance, fig. 9 shows the results of adjusting the threshold filter in *GeRoMeSuite* for the company example. The graph shows the results for the children matcher. The best results are those already displayed in fig. 5. We varied the thresholds with steps of 0.05 in an interval from 0.3 to 0.95. The optimal values are reached at a threshold value of 0.85 and 0.90, respectively. However, the results for a threshold of 0.8 or 0.95 were not considerably worse. For most of the examples we have tested, the best result in terms of f-measure and overall value was produced with threshold values of 0.75 to 0.95.

The stability of the result quality of our matching system with respect to the configuration options is important if "real" matching problems have to be solved, i.e. without having a reference mapping to figure out the best configuration parameters. Using the configuration mentioned above for matching ontologies with XML Schemas, we are confident that the quality of the result is very close to the best result that could be produced with *GeRoMeSuite*.

5.3 Effect of Traversals on the Quality

In section 4, we explained our approach of providing different structural views on the same model. Using traversal strategies, structural matchers can be applied to these different structures, which has an effect on the matching results.

Fig. 10 displays the effect on the quality of results of the same Similarity Flooding matcher which uses different traversal strategies to compute its propagation graph. These are results of matching the geographical example. All matchers had identical configurations except for the traversal strategy. Furthermore, the same filters have been applied to all matcher results. The traversal strategies used were *Association* (A), *Structure* (S) and *Types* (T) and variations of these traversals that omit anonymous model elements from the graph (AV, SV, TV). It can be seen that different graph structures which induce different propagation graphs result in different morphism quality. However, the impact of different traversals on the match result is less than expected. This is probably due to the fact that the similarity of elements in the examples we have tested is mainly determined by the similarity of their labels. Structural similarity has only a small effect on the

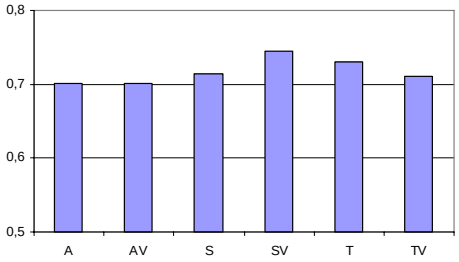

Fig. 10. Performance of Similarity Flooding Using Different Traversal Strategies

match result. This sounds like a counter argument to the idea of structural matchers, but the dominance of string matchers is a particular characteristic of the examples we have chosen. We plan further evaluations on this topic in the context of the Ontology Alignment Evaluation Initiative (`http://oaei.ontologymatching.org/2007/`) which also includes test cases in which only the structural similarity can be used.

5.4 Discussion of the Results

To conclude, for matching ontologies with XML Schemas the children matcher alone or the average of the results of the children matcher and the Similarity Flooding algorithm are a good matcher configuration. The children matcher performed best using the *Association* traversal strategy whereas for similarity flooding the *Structure* traversal strategy was the best choice.

As we implemented only well known schema matching algorithms, the differences in the results of the tools must stem from their internal model representations. For matching ontologies with models from different native modeling languages, the usage of *GeRoMe* as a generic data structure seems to be beneficial. From our experience in the development of the Protoplasm prototype [3], we know that models of different metamodels are represented significantly different when no generic modeling language is used. The graphs represent rather the syntactical structure than the semantics of a model. For example, different labels are used for the edges, and also nodes representing predefined modeling constructs (such as “OWL Class” and “ComplexType”) can have different labels, although the semantics of the model elements is quite similar.

The internal graph structures are not exposed by COMA++, but based on the publications [2,6] and our experience with Protoplasm [3], we can infer that the internal graphs are similar to the graphs shown in fig. 11. The graphs represent the XML schema from section 3 (right part of the figure), and an ontology for the same domain.

These graphs might be easier to understand for humans than the *GeRoMe* representation in section 3. However, as we are dealing with *automatic* schema matching methods, human-readability is not an issue. For a schema matching tool, the graphs contain some problems. First of all, the labels of the edges are different except for the “type” edge. Identical edge labels are for example an important requirement for the Similarity Flooding algorithm as its main data structure, the propagation graph, is build

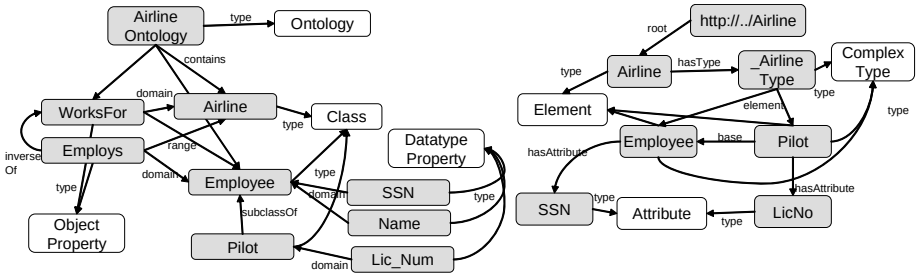

Fig. 11. Graph representation of an ontology and an XML schema

according to identical edge labels in the two graphs. If the labels are different, then the propagation of similarity values to neighboring nodes does not work.

Furthermore, the structure of the graphs is different although the same domain is represented. For example, the association between Airline and Employee/Pilot is not directly visible in the XML schema. Thus, the structural similarity will be considered as very low.

6 Conclusion

We implemented a schema matching subsystem for our holistic model management system *GeRoMeSuite* [14] to match models represented in different metamodels. Our results show the usefulness of our generic metamodel *GeRoMe* for generic model management tasks. The matcher returned comparatively good results when matching models represented in different modeling languages. The comparison with COMA++, another matching system capable of matching XML schemas and OWL ontologies, has shown that *GeRoMeSuite* achieved better results in all test cases. Our system provides several algorithms for element level and structure level matchers; these basic matchers can be combined in a very flexible way which enables the definition of arbitrary matcher combinations. The evaluation has shown that the combination of matchers leads often to better results than the individual matchers.

Furthermore, our matching system is quite stable with respect to different scenarios and configuration options. Using a reasonable combination of matchers and a high threshold value produces a result which is close to the best result that can be achieved with our matcher. Thus, the application of our system to new scenarios can use a standard configuration. Therefore, the user does not need to have a deep understanding of the system, and can still expect a good result of the matching system.

Our results suggest that the usage of a generic metamodel can improve the performance even of model management operators that do not rely on detailed semantics of metamodel constructs, such as the Match operator. Algorithms for matching models are usually interested only in properties of individual nodes, such as labels or types, and in the abstract graph structure of the model. However, the unification of structure that

comes along with using a generic metamodel improves their results. Our matching system provides also various traversal strategies for models, and is not restricted to one graph representation of the model. Depending on the structural information available, the user can choose an appropriate traversal strategy (e.g. IsA hierarchy, associations).

In the near future we plan to improve the usability of our matcher application. Improving the usability and visualization in matching systems is itself an active research area [25]. Automated focussing of matching elements, collapsing and expanding trees when exploring a mapping are already included in our current prototype. We also plan to provide algorithms for sorting the children of tree nodes such that the number of line crossings is minimized. This would highly increase the readability of morphisms.

As our matching subsystem is very easily extendable, it forms a thorough basis for further research on schema and ontology matching. Therefore, we will also implement and evaluate more and new matcher components, and apply them to other homogeneous and heterogeneous matching scenarios. The currently implemented matching components are general purpose components that we can apply to any kind of models. Our next steps include the definition of special purpose matcher components that exploit the characteristics of particular metamodels, e.g. OWL ontologies.

Acknowledgements: This work is supported by the DFG Research Cluster on Ultra High-Speed Mobile Information and Communication (UMIC, `http://www.umic.rwth-aachen.de`).

References

1. 2006.
2. D. Aumueller, H. H. Do, S. Massmann, E. Rahm. Schema and ontology matching with COMA++. *Proc. SIGMOD Conf.*, pp. 906–908. ACM Press, 2005.
3. P. A. Bernstein, S. Melnik, M. Petropoulos, C. Quix. Industrial-Strength Schema Matching. *SIGMOD Record*, **33**(4):38–43, 2004.
4. S. Castano, V. D. Antonellis, S. D. C. di Vimercati. Global Viewing of Heterogeneous Data Sources. *IEEE Transactions on Knowledge and Data Engineering*, **13**(2):277–297, 2001.
5. H. H. Do, S. Melnik, E. Rahm. Comparison of Schema Matching Evaluations. *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web Services, and Database Systems*, pp. 221–237. Springer-Verlag, 2003.
6. H. H. Do, E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. *Proc. 28th Intl. Conf. Very Large Data Bases (VLDB)*, pp. 610–621. 2002.
7. J. Euzenat, P. Valtchev. Similarity-based ontology alignment in OWL-Lite. *Proc. 16th European Conference on Artificial Intelligence (ECAI-04)*, pp. 333–337. 2004.
8. M. A. Hernández, R. J. Miller, L. M. Haas. Clio: A Semi-Automatic Tool For Schema Mapping. *Proc. ACM SIGMOD*, p. 607. 2001.
9. W. Hu, G. Cheng, D. Zheng, X. Zhong, Y. Qu. The Results of Falcon-AO in the OAEI 2006 Campaign. *Intl. Workshop on Ontology Matching (OM-2006)* [1].
10. ISO/IEC. Information technology – Information Resource Dictionary System (IRDS) Framework. *International Standard ISO/IEC 10027:1990*, 1990.
11. M. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, **14**:491498, 1995.
12. Y. Kalfoglou, M. Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, **18**(1):1–31, 2003.

13. D. Kensche, C. Quix, M. A. Chatti, M. Jarke. *GeRoMe*: A Generic Role Based Metamodel for Model Management. *Journal on Data Semantics*, **VIII**:82–117, 2007.
14. D. Kensche, C. Quix, X. Li, Y. Li. *GeRoMeSuite*: A System for Holistic Generic Model Management. *Proc. 33rd Int. Conf. on Very Large Data Bases*. 2007. To appear.
15. D. Kensche, C. Quix, Y. Li, M. Jarke. Generic Schema Mappings. *Proc. 26th Intl. Conf. on Conceptual Modeling (ER'07)*. 2007. To appear.
16. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, **10**:707710, 1966.
17. Y. Li, J. Li, D. Zhang, J. Tang. Result of Ontology Alignment with RiMOM at OAEI06. *Intl. Workshop on Ontology Matching (OM-2006)* [1].
18. J. Madhavan, P. A. Bernstein, E. Rahm. Generic Schema Matching with Cupid. *Proc. 27th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 49–58. Rome, Italy, 2001.
19. S. Massmann, D. Engmann, E. Rahm. COMA++: Results for the Ontology Alignment Contest OAEI 2006. *Intl. Workshop on Ontology Matching (OM-2006)* [1].
20. S. Melnik, H. Garcia-Molina, E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. *Proc. 18th Intl. Conference on Data Engineering (ICDE)*, pp. 117–128. IEEE, 2002.
21. N. F. Noy. Semantic Integration: A Survey Of Ontology-Based Approaches. *SIGMOD Record*, **33**(4):65–70, 2004.
22. Object Management Group. Common Warehouse Metamodel (CWM), version 1.0. *Spezifikation*, Feb 2001.
23. C. Quix, D. Kensche, X. Li. Generic Schema Merging. J. Krogstie, A. Opdahl, G. Sindre (eds.), *Proc. 19th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'07)*, LNCS, pp. 127–141. Springer-Verlag, 2007.
24. E. Rahm, P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, **10**(4):334–350, 2001.
25. G. G. Robertson, M. Czerwinski, J. E. Churchill. Visualization of mappings between schemas. *Proc. Conf. on Human Factors in Computing Systems (CHI)*, pp. 431–439. ACM, 2005.
26. P. Shvaiko, J. Euzenat. A Survey of Schema-Based Matching Approaches. *Journal on Data Semantics*, **IV**:146–171, 2005. LNCS 3730.
27. G. Stoilos, G. B. Stamou, S. D. Kollias. A String Metric for Ontology Alignment. *Proc. 4th Intl. Semantic Web Conference (ISWC)*, *LNCS*, vol. 3729, pp. 624–637. Springer, 2005.
28. W. Winkler. The state record linkage and current research problems. *Tech. rep.*, Statistics of Income Division, Internal Revenue Service Publication, 1999.