Industrial-Strength Schema Matching

Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, Christoph Quix

Microsoft Research
Redmond, WA, U.S.A.
{philbe, melnik}@microsoft.com, mpetropo@cs.ucsd.edu, quix@informatik.rwth-aachen.de

Abstract

Schema matching identifies elements of two given schemas that correspond to each other. Although there are many algorithms for schema matching, little has been written about building a system that can be used in practice. We describe our initial experience building such a system, a customizable schema matcher called Protoplasm.

1 Schema Matching

Most systems-integration work requires creating mappings between models, such as database schemas, message formats, interface definitions, and user-interface forms. As a design activity, schema mapping is similar to database design in that it requires digging deeply into the semantics of schemas. This is usually quite time consuming, not only in level-of-effort but also in elapsed-time, because the process of teasing out semantics is slow. It can benefit from the development of improved tools, but it is unlikely such tools will provide a silver bullet that automates all of the work. For example, it is hard to imagine how the best tool could eliminate the need for a designer to read documentation, ask developers and end-users how they use the data, or review test runs to check for unexpected results. In a sense, the problem is AI-complete, that is, as hard as reproducing human intelligence.

The best commercially-available model mapping tools we know of are basically graphical programming tools. That is, they allow one to specify a schema mapping as a directed graph where nodes are simple data transformations and edges are data flows. Such tools help specify: a mapping between two messages, a data warehouse loading script, or a database query. While such graphical programming is an improvement over typing code, no database design intelligence is being offered. Despite the limited expectations expressed in the previous paragraph, we should certainly be able to offer some intelligence — automated help, in addition to attractive graphics.

There are two steps to automating the creation of mappings between schemas: schema matching and query discovery. Schema matching identifies elements that correspond to each other, but does not explain how they correspond. For example, it might say that FirstName and LastName in one schema are related to Name in the other, but not say that concatenating the former yields the latter. Query discovery picks up where schema matching leaves off. Given the correspondences, it obtains queries to translate instances of the source schema into instances of the target, e.g., using query analysis and data mining [5].

This paper focuses on schema matching. There are many algorithms to solve it [8]. They exploit name similarity, thesauri, schema structure, instances, value distribution of instances, past mappings, constraints, cluster analysis of a schema corpus, and similarity to standard schemas. All of these algorithms have merit. So what we need is a toolset that incorporates them in an integrated package. This is the subject of this paper.

The published work on schema matching is mostly about algorithms, not systems. This algorithm work is helpful, offering new ways to produce mappings that previous algorithms were unable to find. However, all of the published algorithms are fragile: they often need manual tuning, such as setting thresholds, providing a thesaurus, or being trained on examples; even after tuning, it is easy to find schemas that the algorithms do not map correctly; and many of them do not scale to large schemas.

COMA is the first work to address engineering issues of a schema matching system [1]. Its architecture offers multiple schema-level matchers and a fixed process to combine their results. Matchers exploit linguistic, datatype, and structural information, plus previous matches, to produce similarity matrices forming a cube. The cube is aggregated to a matrix. Then particular similarity values are selected as good match candidates, which are combined to a single value. This process is executed for whole schemas or for two schema elements, and is repeated after the user provides feedback. Experimentation showed that finding good combinations of matchers depends on characteristics of the schemas being matched, demonstrating that customizability of the combination process is crucial.

Inspired by COMA's approach, we felt it was important to push further toward building an industrial-strength schema matcher, one that avoids fragility problems, is customizable for use in practical applications, and extends the range of matching solutions being offered.

We believe fragility is inherent in the problem. To mitigate it, we need a system that can exploit the best algorithms and is customizable by a human designer. Designers use all of the techniques found in specific schema matching algorithms: name similarity, thesauri, common schema structure, overlapping instances, common value distribution, re-use of past mappings, constraints, similarity to standard schemas, and common-sense reasoning. A system should use all of these techniques too.

Customizability is needed for several reasons. First, a user can improve a schema matching tool by selecting particular techniques and combining them in a way that works best for the types of schemas being matched. For example, if a version ID is prepended to element names in evolving schemas, then a user may want to delete the ID before applying any matcher**.** The degree of such customizability depends on the user's sophistication. An end-user who is matching two schemas wants a limited range of options to choose from. By contrast, a sophisticated user, such as an application vendor, wants to customize the tool to work well when matching its schemas to those of other parties. A second reason for customizability is to control an algorithm's scalability, e.g., by trading off response-time for the quality of the result. A third reason is extensibility, meaning that new techniques can be easily added to the tool. This helps sophisticated users, and researchers who want to experiment with new or modified algorithms.

We therefore embarked on a project to build such a customizable schema matcher, one that could be used in commercial settings. This paper describes our early experience in developing this tool, called Protoplasm (a PROTOtype PLAtform for Schema Matching). In addition to arguing for the importance of research on industrial-strength schema matching, we offer two main contributions: (i) a new architecture for schema matching, which includes two new internal schema representations, interfaces for operators that comprise a matching algorithm, and a flexible approach to combining operators; and (ii) experience in tuning the prototype for scalability and using its customization features to add another algorithm. These are covered in Sections 2 and 3 respectively. We close in Section 4 with a discussion of future work.

2 Architecture

Figure 1 presents the three-layer architecture of Protoplasm. The bottom layer consists of two supporting structures. The *Schema Matching Model Graph* (*SMM Graph*) is the internal representation of input schemas: a rooted node- and edge-labeled directed graph. The *Similarity Matrix* holds correspondences between two SMM graphs. The middle layer defines a set of *Operator Interfaces* that declare the inputs and outputs of generic schema-matching tasks such as import, transformation and export of schemas, creation and manipulation of similarity matrices, and match. Protoplasm offers implementations of these interfaces. Custom ones are easy to plug in, which is one dimension of the extensibility of the platform. The other dimension is the *Strategy Scripts* in the top layer, which implement match algorithms consisting of interconnected operators. Each script combines a set of operator implementations by passing SMM graphs and similarity matrices from one to another. Ideally, strategies would be designed using a *Graphical Interface* that generates the strategy script from its graphical representation.

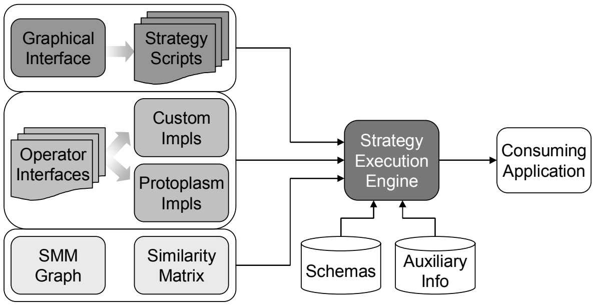


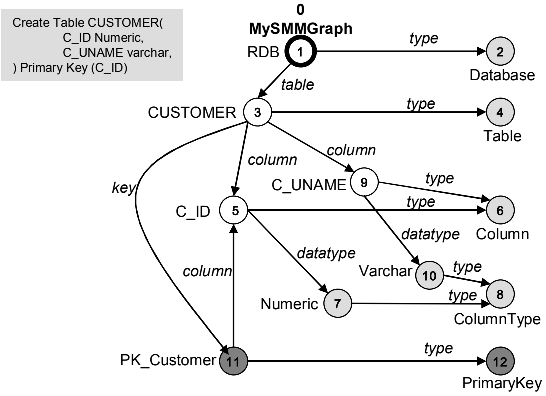
Figure 1 Protoplasm Architecture

The *Strategy Execution Engine* takes as input a strategy script, the Protoplasm-provided and custom operator implementations, and SMM graphs and similarity matrices. During the execution, the engine accesses schema repositories and other auxiliary information, e.g., a thesaurus or glossary, used by implementations of operators in the strategy script. The execution of the script results in a similarity matrix exported in a format compatible with an external *Consuming Application*.

2.1 Representing Models

The input schemas can be in any meta-model, e.g., SQL DDL, ODMG or XML Schema Definition (XSD) Language. Since no existing meta-model is expressive enough to subsume all others, Protoplasm defines SMM Graphs as a meta-model-independent representation. An SMM graph is a rooted, node- and edge-labeled directed graph, where each node and edge has a unique identifier. Figure 2 shows an example SMM Graph describing the SQL DDL in the shaded box that defines a relational table. The root (i.e., `id`=1) is indicated by a thick circle. The graph identifier (i.e., `id`=0) and label are in boldface above the root. The database, table and column names are denoted by white circles, typing information by light gray circles, and constraints (e.g., primary key) by dark gray circles. SMM is a variation of the Object Exchange Model (OEM) [7], a data model defined for semistructured data, thus inheriting its simplicity and self-describing nature.

Protoplasm builds on widely-used XML technologies to implement SMM graphs. An XML syntax is defined to describe and store SMM graphs in secondary memory (e.g. disk), an extended XML parser is employed to load them into main memory, a variation of DOM is used to enable programmatic navigation and editing, and an XPath engine is modified to execute simple queries against SMM graphs. The XML syntax describing SMM graphs uses the node ids to establish the edges between them and is validated using an XML schema that makes use of the `key` and `keyref` definition elements. We used .NET tools for the XPath engine, XSD validator, etc.



```
Create Table CUSTOMER(
        C_ID Numeric,
        C_UNAME varchar,
) Primary Key (C_ID)
```

Figure 2 Example SMM Graph

DOM is not used "as-is" for navigation since it does not provide a way to navigate gracefully across `keys` and `keyrefs`. We resolve this issue by deriving from DOM an object model for SMM, called *Graph Document Object Model* (*GDOM*), which specializes DOM as follows: (i) GDOM graph, nodes and edges are DOM elements with `label` and `id` properties only; (ii) nodes have a list of outgoing edges as children; (iii) edges have their start node as parent and their end node as their only child.

The GDOM main memory representation of SMM graphs is constructed directly during parsing. A DOM parser was extended to construct GDOM elements and establish their connections based on the node ids.

GDOM also inherits the editing capabilities of DOM. But in the case of SMM graphs, DOM editing capabilities are more powerful than needed and allow the construction of invalid SMM graphs. For this reason, Protoplasm provides a GDOM wrapper interface that exports only the editing methods that result in valid SMM graphs.

GDOM documents cannot be queried conveniently by XPath expressions, since XPath does not provide a way to navigate across `keys` and `keyrefs`. Instead, GDOM documents are queried by XGPath, a variant of XPath that we implemented to overcome this limitation. XGPath defines two macros, `node()` and `edge()`, to recognize nodes and edges when computing a transitive closure, and allows only `child` and `descendant` axes. For example, query Q1 below finds all nodes in the graph. Q2 finds all nodes having at least one outgoing edge.

```
(Q1) /descendant::*[g:node()]
(Q2) /descendant::*[g:node()][*[g:edge()]]
```

2.2 Similarity Matrix

A *Similarity Matrix* holds the degree of similarity between items (nodes and/or edges) of two SMM graphs. As in COMA, it is the main integration point for combining results from different match algorithms. The rows and columns of a similarity matrix represent lists of items from the source and target graphs, respectively. The lists can be constructed by running an XGPath query over the graphs. Each cell of the matrix holds the ids of the target and source items and the similarity value between them.

2.3 Operators

Protoplasm declares a set of operator interfaces to carry out individual schema matching tasks. SMM graphs, similarity matrices and cells are the possible input and output types. These operators are organized in the following three groups according to their functionality:

Import, Transform and Export. These handle the starting and ending stages of a match algorithm. One *Import* operator interface takes as input a schema expressed in some meta-model and outputs a corresponding SMM graph. Another import interface is declared to load an existing SMM graph or similarity matrix. Implementations of the import interface are meta-model-specific.

A *Transform* operator interface applies a transformation to an input SMM graph, returning a transformed one. For example, an imported SMM graph might need to be transformed to encode schema constraints differently for a particular match algorithm. Implementations of the transform interface are specific to the match algorithm used.

An *Export* operator interface translates a similarity matrix into a format suitable for the consuming application, e.g., BizTalk Mapper [4]. Since each application has its own format for representing mappings, each implementation of the export interface is application-specific.

Matrix Creation and Manipulation. This group consists of seven operator interfaces manipulating matrices holding match results.

1. The *CreateSM* operator interface creates a new similarity matrix given two input SMM graphs.
2. The *CellToSM* operator interface takes as input a cell of a similarity matrix and generates a nested similarity matrix based on some analysis. For example, when applied on a cell holding two node labels, *CellToSM* can generate a nested similarity matrix by decomposing the cell's labels. Rows and columns of the nested matrix are the components of the node labels, which can now be matched individually.
3. The inverse operator interface is *AggSMToCell*. Given a cell c that was used to generate a similarity matrix sm using *CellToSM*, it aggregates the similarity values of sm and assigns the aggregation to c.
4. The *MergeSMs* operator interface merges a set of input heterogeneous similarity matrices into a single output one by aggregating similarity values across matrices. For example, multiple matching techniques are applied to generate different similarity matrices, which are then merged into a single matrix.
5. Analogously, the *MergeCells* operator interface merges the similarities of a set of input cells into a single one.
6. The *TraverseSM* operator interface creates a cursor that iterates over the cells of a similarity matrix. The traversal order can be matrix-specific, e.g., rows-first, or graph-specific, e.g., bottom-up.
7. The *FilterSM* operator interface takes as input a similarity matrix whose values are in the range [0,1] and

outputs a similarity matrix whose values are 0 or 1. Deleting cells whose values are below a threshold is an example of a filter implementation.

Match. The *Match* operator interface takes as input a cell with or without a given similarity value and calculates a new one based on the cell's items.

2.4 Scripts

For customizability, implementations of operator interfaces can be combined in many different ways to execute a match algorithm. A given combination, called a *strategy*, consists of operators and a control flow that tells how the output of operators is passed as the input to other operators. In the current implementation, strategies are coded using the procedural language C# and are compiled and executed by the execution engine. An example strategy is shown in Figure 3 as it would be displayed by a graphical interface. Operator interfaces are indicated by icons labeled by the name of the implementation. Lines connecting icons pass SMM graphs, similarity matrices, or cells from one operator to another. The "Level" labels indicate if the operators execute against whole similarity matrices or individual cells.

The strategy starts by executing two Import operators that result in two SMM graphs representing two XML schemas. Then two CreateSM operators create a similarity matrix M_1 consisting of all nodes of the two SMM graphs, and an M_2 consisting of only the internal ones. M_1 will be used to hold the linguistic similarity of all nodes, while M_2 will hold the structural similarity of all internal ones.

First, a TraverseSM operator iterates over the cells of M_1 rows-first. Then a CellToSM operator generates a similarity matrix M_3 for the currently traversed cell of M_1 by tokenizing the labels of its items based on the camel case naming convention. Another TraverseSM operator then iterates over the cells of M_3 and applies a Match operator based on stems. Subsequently, an AggSMToCell operator takes the average of the similarity values in M_3 and places it in the currently traversed cell of M_1.

Another TraverseSM operator iterates over M_2 bottom-up and applies to each cell a Match operator that calculates the structural similarity of two internal nodes based on the linguistic similarity their leaves. The two resulting matrices M_1 and M_2 are then passed to a MergeSMs operator that merges them into one. The similarity values of common cells are merged based on a weighted formula. A FilterSM operator filters out poor similarity values below a threshold. Finally, an Export operator transforms the resulting matrix into a BizTalk Map.

Many published algorithms use similarity matrices in a way similar to Protoplasm [1,2,3,7]. However, except for COMA, none are designed as an open integration platform in which new algorithms and heuristics can be easily incorporated. COMA too is limited in that it combines the result of match algorithms by taking a linear combination of the similarity matrices they produce. Protoplasm offers more flexible combinations. For example, the strategy in Figure 3 pipelines matchers; the linguistic matcher passes its output to the structural matcher. Furthermore, Protoplasm is designed for customizability and adaptability: data structures and operators can be easily extended with customized implementations.

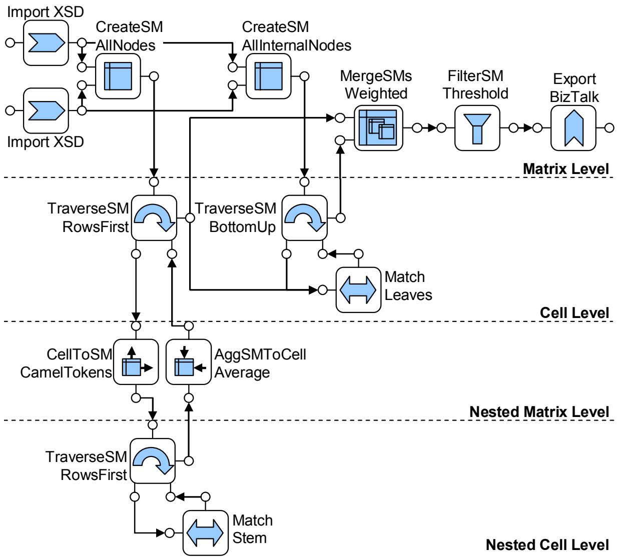


Figure 3 Example Strategy

3 Experience

3.1 Scalability

The first prototype of Protoplasm was applied to a real-world problem to match two versions of an EDI schema, expressed in XML Schema. The old version contained 340 schema elements (element types and attributes) and the new version contained 500. As a version number was encoded into the element names, a direct name match was not possible, even for elements that did not change.

We used a simplified version of the Cupid algorithm [2] to match the schemas, which resembles the strategy in Figure 3. As this was the first real-world test of the prototype, we were not surprised to find scalability problems: it required a few hundred MB of main memory and took over an hour to construct the first linguistic similarity matrix (and several more matrices were needed for type, structural and combined similarities).

Our profiling showed that the first step of the algorithm, which transforms the schema graph into a tree, multiplied the number of elements by a factor of 6. Hence, each matrix had 2K×3K elements; at 20 bytes/cell, that is 120 MB/matrix. Many match algorithms require the schema representation to be a tree. So Protoplasm provides an operator to transform a graph into a tree by duplicating nodes that have multiple incoming edges. In the example problem, only one node had more than two incoming edges. But the operator is applied recursively to all nodes, resulting in a much bigger graph than expected. We optimized the graph-to-tree operator so that it exploits semantic information. For example, it removes certain edges that do not provide useful information for schema matching to avoid duplicating nodes. In the example, every element type *t* had an incoming edge from the root, meaning that *t* is a component of the schema. These edges could be removed if *t* was referenced by another type. Using this optimization, the schema tree for this example has about the same number of nodes as the graph.

Our initial linguistic matcher compared each element of schema 1 with each element of schema 2, an O(n^2) algorithm. After seeing the scalability implication, we re-implemented it using a strategy similar to hash join: the names (or tokens or ngrams) of schema 1 are inserted into a hash table. Then, for each element of schema 2 we look up the matching elements of schema 1 in the hash table. This reduces the complexity to O(n) and the time to compute a linguistic matrix with 100x1000 cells from 75 to 3 seconds.

Another optimization strategy is to cache intermediate results, e.g., when retrieving the leaves under a schema element. This requires more main memory, but reduces the execution time by a factor of 2 or 3. Using all of the above techniques, we reduced the execution time for this example from several hours to less than a minute.

Scalability issues are also discussed in [6]. However, this match algorithm was tuned for a particular, much larger matching problem and was implemented in SQL.

3.2 Revised Object Model

In our experiments we encountered several issues with our data model and query language. The representation of the schema graph in GDOM enabled the reuse of existing XML technologies that developers are familiar with but also had significant main memory overhead. Furthermore, profiling showed that much time was spent in evaluating XGPath expressions, especially in the structural matching phase, where we frequently navigate between internal nodes to their leaf nodes.

We concluded that a more flexible and efficient object model should be offered that enables the usage of index structures for navigation and simplifies the integration with existing object models. The revised object model that we developed provides a lightweight representation of the graph using *delegate functions*, a feature of the .NET framework that is similar to function pointers in C/C++. Existing object models, such as XML Schema Object Model (SOM) in .NET, are wrapped by implementing delegate functions that enable navigational access to the underlying object graph. We migrated Protoplasm to the new lightweight object model. GDOM (and the navigation via XGPath expressions) is now the default graph implementation when no native object model is available. GDOM's graph navigation primitives (e.g., retrieving the "name"-edge for a given node), which were originally implemented using XGPath expressions, can be overridden by optimized delegate functions. Graph access performance was thereby improved by a factor of 2 to 3.

A major advantage of the revised object model is that the existing code base can be reused by wrapping native object models. The delegate functions can make graph navigation very efficient by utilizing index structures or access methods of the native object model.

3.3 Extensibility

As mentioned in Section2.4, Protoplasm can be extended by adding schema matching scripts or customized data structures and operators. To validate the extensibility of Protoplasm, we implemented another schema matching strategy, Similarity Flooding (SF) [3]. We chose this algorithm because it is significantly different from the ones we designed for: it is based on a different data structure (the propagation graph) and uses a fixpoint computation.

By reusing the Protoplasm infrastructure, we were able to implement this algorithm in two days. We reused data structures for similarity matrices and schema graphs, operators to import and transform schemas, and a match operator to compute the initial similarity values. But we needed to implement a new data structure for the propagation graph, which was the most time-consuming part. The amount of code and total time to design and implement the algorithm were reduced by about half, compared to implementing it from scratch. Given our previous experience with memory and performance problems, we chose a very compact representation for the propagation graph. The propagation values are stored in an array. A special indexing technique provides direct access to the corresponding schema elements and cells in the similarity matrix, so navigation between the propagation graph and the similarity matrix can be done in constant time. This data structure also saves about 60% of main memory, compared to a standard implementation of the graph.

We ran several experiments using this implementation of SF. As shown in [3], we verified that SF is strong in detecting similar structures. For example, schemas that had identical structure but different element names (because of different languages) were matched exactly. However, this feature of SF causes problems if a schema is self-similar, i.e., a complex structure repeats within the schema (e.g., address). In this case, SF matches each copy of the repeating structure of one schema to all copies in the other. It may be possible to address this limitation by developing a matching strategy that filters an ambiguous match result by running SF on each individual schema to identify self-similar structures

3.4 Lessons Learned

One important lesson is that performance is an issue, even though we are "just" handling metadata. The real-world example in Section 3.1 is still relatively small; schemas with several thousands elements are common in e-business applications. Since schema matching is a semi-automatic task, efficient implementations are required to support interactive user feedback.

Customizability and flexibility proved to be important. Schema matching problems differ along various dimensions, such as naming conventions, modeling styles (expressing the same thing in different ways), degrees of similarity between schemas, and user requirements. Thus, a single solution is unlikely to perform equally well across all matching problems. Therefore, easy customization and adaptation of a basic strategy is necessary.

The graphical design of strategy scripts (see Figure 3) might still be too complex for an administrator or database designer lacking a deep understanding of schema matching issues. Higher-level operators are required that work on complete graphs (as opposed to individual nodes) and provide functions to match schema elements by their names, types, or structure. These operators can be used as the basic building blocks of a schema matching strategy.

The need for higher-level operators coincides with our experience in optimizing the operators. Efficient implementation of the operators is only possible if they are executed at the graph level. A simple iteration over all cells of a similarity matrix (as in Figure 3) is always $O(n^2)$.

4 Future Work

Since schema matching algorithms are inherently fragile, a customizable schema matcher needs to include multiple easily-customizable scripts. One challenge is to define a few such scripts that cover the space of useful matchers and offer a small set of simple customization choices.

An important capability of a schema matcher is reuse of previously-developed mappings. Often, a reusable mapping is between sub-schemas of the schema matcher's inputs. The combinatorics of finding reusable mappings and applying them to a given mapping problem can be daunting. There has been some work on reuse but much more is needed [1].

A customizable schema matcher could become a large, complex system. In addition to the current features of Protoplasm, it could include the following subsystems:

- Natural language processing (NLP) – glossaries and schema documentation are analyzed to produce thesauri that the schema matcher uses to identify synonyms and homonyms. NLP is also useful to determine the similarity of short phrases that describe elements of the two schemas.
- Machine learning – a machine learner is used to capture and reuse validated correspondences.
- Data mining – algorithms for comparing the values or distributions of instances of different elements to decide if they are similar.
- Semantic analysis of mappings – since schemas are complex semantic structures, inferencing over them may help identify redundant or inconsistent matches.
- Constraint solver – given similarity scores between pairs of schema elements, a best mapping is picked that satisfies given mapping constraints (e.g., each target element can connect to at most one source element).

Moreover, there is a need for a clever and flexible user interface (UI) to display match results. Users and tool designers tell us that the problem of UI clutter in schema matching tools is at least as important as the tool's lack of intelligence. When matching two large schemas, it is hard to find your way around, remember where you have been, explore several alternative matches concurrently, and leave a trail of annotations that captures what you learned.

One challenge in using the above technologies is coping with the fixed interfaces of large existing subsystems that implement them. For example, some NLP systems can produce a semantic network for individual sentences, but leave it to the caller to transform the network into a thesaurus. Machine-generated and hand-crafted thesauri often lack similarity scores, which are needed by most match algorithms. Some learning algorithms capture past matches in an executable learning network, not as a data structure that can be combined with the output of other algorithms. Most matching algorithms are batch-oriented, matching a schema at a time, whereas some tools need an incremental matcher that the user can steer, using each match decision to influence the choice of later ones.

Another challenge is coping with the large size of the components to be integrated, such as NLP, machine learning, and constraint solving systems. Most were designed for stand-alone use. Integrating them into one platform will undoubtedly generate software engineering problems.

Given the size and complexity of such a system, it is unlikely that a company can afford to build more than one of them. Thus, it must be reusable in many kinds of tools, across many different data models and natural languages. One goal of building such a system is to learn the right requirements by reusing it in all of these contexts.

References

1. Do, H. H., and E. Rahm: COMA - A System for Flexible Combination of Schema Matching Approaches. VLDB 2002: 610-621.
2. Madhavan, J., P.A. Bernstein, E. Rahm: Generic Schema Matching with Cupid. VLDB 2001: 49–58.
3. Melnik, S., H. Garcia-Molina, E. Rahm: Similarity Flooding - A Versatile Graph Matching Algorithm. ICDE 2002.
4. Microsoft Corporation: BizTalk Mapper. http://www.microsoft.com/technet/biztalk/btsdocs
5. Miller, R.J., L. Haas, M.A. Hernández: Schema Mapping as Query Discovery. VLDB 2000: 77–88.
6. Mork, P. and P. A. Bernstein: Adapting a Generic Match Algorithm to Align Ontologies of Human Anatomy. ICDE 2004: 787-790.
7. Papakonstantinou Y., H. Garcia-Molina, J. Widom: Object Exchange Across Heterogeneous Information Sources. ICDE 1995.
8. Rahm, E., P.A. Bernstein: A Survey of Approaches to Automatic Schema Matching. VLDB J. 10(4), 2001.