Industrial Strength Schema Matching

Philip A. Bernstein[1]
Microsoft Research

The problem of creating mappings between schemas is an unavoidable and time-consuming step in the design of many types of large-scale commercial applications: in transaction processing and enterprise application integration, to help map messages between different XML formats; in data warehouses, to map data sources into warehouse schemas; and in web portals, to identify points of integration between heterogeneous databases. This position paper gives a brief summary of the state-of-the-art of schema matching and recommends platform development, rather than algorithm development, as the direction most likely to yield a significant improvement.

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. Like database design, it can benefit from the development of improved tools, but it is unlikely such tools will provide a silver bullet that automates most of the work. For example, it is hard to imagine how the best tool could eliminate the need for a database designer to read documentation, browse data instances, and talk to application developers and end users about how they use the data. In a sense, the problem is AI complete, that is, as hard as reproducing human intelligence.

The best commercially-available schema mappings tools we know of are basically graphical programming tools. That is, they allow one to specify a schema mapping as a directed graph whose nodes are simple data transformations and whose edges are data flows. Such tools help specify a mapping between two messages, a data warehouse loading script, or a database query. While this sort of graphical programming is a significant improvement over simply typing code, there's no database design intelligence being offered. Despite the limited expectations expressed in the previous paragraph, we should certainly be able to offer such intelligence -– automated help, not just attractive graphics.

There have been two lines of research focused on partially automating the creation of mappings between schemas: schema matching and query discovery. *Schema matching* (also called *mapping discovery*) identifies elements that correspond to each other, but does not say much about the nature of the correspondence. For example, it might say that FirstName and LastName in one schema are related to Name in the other, without saying that the former are concatenated to obtain the latter. *Query discovery* picks up where mapping discovery leaves off. Given the correspondences, it obtains queries for translating instances of the source schema into instances of the target.

[1] Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399.
email: philbe@microsoft.com

There are numerous algorithms for schema matching [5]. They exploit name similarity, thesauri, common schema structure, common instances, common value distribution of instances, re-use of past mappings, constraints, cluster analysis of large schema sets, and similarity to standard schemas (i.e., elements similar to the same standard element are similar to each other). Query discovery includes query analysis and data mining techniques for recommending joins, selects, projects, and perhaps someday to suggest instance-level transformations (such as the concatenation of FirstName and LastName above).

One advanced project on query discovery is Clio at IBM [4]. Clio offers a toolset that incorporates many algorithms in an integrated package. We believe the same must be done for schema matching, which is the subject of the rest of this paper.

With the exception of COMA[1], the published work on schema matching has been about algorithms, not systems. Much of this work has been helpful, offering new algorithms that can produce mappings that previous algorithms were unable to find. However, all of the published algorithms are fragile, in several senses: first, they often have magic numbers that need to be calibrated and are sensitive to the examples that are used to calibrate them; second, it is easy to find schemas that the algorithms are unable to correctly map; and third, many of them are not scalable to large schemas. Regarding the latter point, we were recently involved in developing a prototype that matched two large ontologies of human anatomy, each of which consists of approximately 50K classes and a million relationships. For such large schemas, an $O(n^2)$ algorithm can take day or two to execute. A flooding algorithm that iterates the $O(n^2)$ algorithm till it converges could take weeks.

To make progress, we believe it's important to build an industrial strength schema matcher, one that avoids these fragility problems and is customizable for use in practical applications. We believe fragility is inherent in the problem. To mitigate the problem, what is needed is not a better algorithm, but rather an architecture for a system that can exploit all of the best algorithms and that can be controlled by a human designer. Human designers ordinarily use all of the techniques that have been applied in specific schema matching algorithms: name similarity, thesauri, common schema structure, common instances, common value distribution of instances, re-use of past mappings, constraints, similarity to standard schemas, plus common sense reasoning. It is unlikely that a system that does less will be regarded as satisfactory by such users. Human control over the algorithm is also needed for scalability, so the user can make a tradeoff between response time and the quality of the result. To do all of this will require a big system consisting of many large components.

The following kinds of algorithms are used in schema matching:

- Natural language processing (NLP) – glossaries and schema documentation is analyzed to produce thesauri that are used by the schema matcher to identify synonyms and homonyms. NLP is also useful for determining the similarity of short phrases that describe elements of the two schemas.

- Lexical analysis – compound names are normalized into multi-word phrases by decomposing them based on punctuation, grammar, and dictionaries.
- Graph traversal – the schemas are represented as graphs which are traversed in some orderly way to determine correspondences between nodes of the graphs.
- Machine learning – a neural network or the like is used to capture and reuse information about past correspondences.
- Data mining – algorithms for determining whether the values or distributions of instances of different schemas are sufficiently similar to justify concluding that the schema elements describing them are similar.
- Semantic analysis of mappings – schemas are complex semantic structures, not just graphs, so inferencing is required to avoid redundant matches and identify inconsistent matches.
- Constraint solver – given a zero-to-one similarity score between pairs of elements from the two schemas, pick a best mapping that satisfies a give set of mapping constraints (e.g. that 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 present the results of these algorithms to a database designer. Anecdotal evidence from users and tool designers indicates that UI clutter in schema matching tools is currently a bigger problem than the tool's lack of intelligence. When matching two large schemas, it's hard to find your way around, remember where you've been, explore several alternative matches concurrently, and leave a trail of annotations that captures what you learn.

The main integration point to use to combine the above algorithms is a similarity matrix, which is an n-by-m matrix that gives a zero-to-one similarity score to each pair of elements in the two schemas. Using this approach, NLP produces a thesaurus with similarity scores for the synonyms it identifies. Lexical analysis produces similarity scores based on the similarity of word phrases. Some combination of graph traversal and machine learning refines the result of lexical analysis based on graph similarity and past correspondences, respectively. Finally, the constraint solver generates a mapping based on the similarity matrix and the given constraints.

Many published algorithms use similarity matrices in this way [1,2,3,5]. However, with the exception of 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 complete algorithm executions by taking a linear combination of the similarity matrices produced independently by each algorithm. A more flexible way of combining algorithms is desirable. For example, algorithms may be pipelined, such as using lexical analysis as an input to different algorithms or semantic analysis as a post-processing step. Or they may be invoked as subroutines, such as invoking a machine learning algorithm on sub-schemas in the middle of a graph-oriented match.

One challenge in designing an integration platform for schema matching is coping with large existing subsystems some of whose fixed interfaces are inappropriate. For example, some NLP systems are able to

produce some kind of semantic network for individual sentences, but leave it to the caller to figure out how to transform a network into a thesaurus. Such machine generated thesauri, as well as commercially available ones used for word processing, lack similarity scores, which are needed by most matching 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 an entire schema at a time, whereas some tools may need a schema matcher to be incremental, allowing the designer to steer it, using each match decision to influence the choice of later matchers.

Another challenge is coping with the sheer size of the components to be integrated. Systems for NLP, machine learning, and constraint solving are each substantial systems in their own right. Most were designed for stand-alone use. Combining them into an integrated platform will undoubtedly generate software engineering problems.

Given the size and complexity of such a system, it’s unlikely that the industry can support development of more than a few industrial strength versions. Certainly, no company can afford more than one of them. Thus, it must be reusable in the context of many kinds of mapping tools, across many different data models and natural languages. One important goal of building such a system is in learning the right requirements, which means reusing it all of these contexts.

Having spent several years analyzing existing schema matching algorithms and developing new ones, we are now starting to build a prototype for the kind of industrial-strength system described above. We expect it to be a very long road.

References

1. Hong Hai Do, Erhard 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, pp. 49–58.
3. Melnik, S., H. Garcia-Molina, E. Rahm: Similarity Flooding - A Versatile Graph Matching Algorithm. ICDE 2002.
4. Miller, R.J., L. Haas, M.A. Hernández: Schema Mapping as Query Discovery. VLDB 2000, pp. 77–88.
5. Rahm, E., and P. A. Bernstein: A Survey of Approaches to Automatic Schema Matching. VLDB Journal 10(4), 2001.