Schema Mapping Verification: The Spicy Way

A. Bonifati,[1,2] G. Mecca[1], A. Pappalardo[1], S. Raunich[1], G. Summa[1]

[1] Dipartimento di Matematica e Informatica
Università della Basilicata
Potenza – Italy

[2] ICAR – CNR
Rende – Italy

ABSTRACT

Schema mapping algorithms rely on value correspondences – i.e., correspondences among semantically related attributes – to produce complex transformations among data sources. These correspondences are either manually specified or suggested by separate modules called schema matchers. The quality of mappings produced by a mapping generation tool strongly depends on the quality of the input correspondences. In this paper, we introduce the Spicy system, a novel approach to the problem of verifying the quality of mappings. Spicy is based on a three-layer architecture, in which a schema matching module is used to provide input to a mapping generation module. Then, a third module, the mapping verification module, is used to check candidate mappings and choose the ones that represent better transformations of the source into the target. At the core of the system stands a new technique for comparing the structure and actual content of trees, called structural analysis. Experimental results show that, by carefully designing the comparison algorithm, it is possible to achieve both good scalability and high precision in mapping selection.

1. INTRODUCTION

Data integration is considered one of the most intriguing challenges of our era. Given a source repository, S, and a target repository, T, a key step in their integration is that of finding a set of mappings that can be used to transform instances of S into instances of T.

Several systems in recent literature have studied the problem of deriving mappings among sources based on *value correspondences*; each of these correspondences states that an attribute of the target is semantically related to one attribute (or more) in the source, and is usually drawn as a *line* going from the source attribute to the corresponding target attribute. A prominent example of a line–based mapping generation system is Clio [24, 28]. Clio implements a sophisticated mapping algorithm to generate source-to-target transformations, i.e., executable queries capable of translating an arbitrary instance of the source into an instance of the target. The mapping generation algorithm captures all semantical relationships embedded in the source and target schemas, and is guaranteed to produce legal instances of the target with respect to constraints.

It can be seen, however, how a crucial step in the mapping generation process is the discovery of the initial value correspondences. In fact, the quality of the mappings produced by the mapping generation system is strongly influenced by the quality of the input lines: starting from faulty correspondences incorrect mappings are inevitably produced, thus impairing the quality of the overall integration process. To avoid these problems, it is usually assumed that value correspondences are interactively provided to the system by a human expert after carefully browsing the source and target repositories. However, such manual process is very labor-intensive, and does not scale well to medium and large integration tasks.

To alleviate the burden of manually specifying lines, one alternative is to couple the mapping generation system with a *schema matching system*, i.e., a system that automatically or semi-automatically tries to discover matching attributes in a source and target schema. The study of automatic techniques for schema matching has received quite a lot of attention in recent years; for a survey see [29, 11, 30]. Clio itself has been complemented with a companion schema matching module based on attribute feature analysis [25]; this tool may be asked to suggest attribute correspondences to the user.

Unfortunately, schema matching has been recognized as a very challenging problem [18], for which no definitive solution exists: although current schema matching systems perform well in some application categories, in other cases they suffer from poor precision. According to [19], there is no perfect schema matching tool. [19] reports that on a recent benchmark of ontology–matching tasks [1], participating matchers on average achieved 40% precision and 45% recall. Also, even for datasets for which such tools reached higher precision and recall, they still produced inconsistent or erroneous mappings.

As a consequence, outputs of the attribute matching phase are hardly ready to be fed to the mapping generation module, and human intervention is necessary in order to analyze and validate them. It is worth noting that, in general,

EDBT '08 Nantes, France

human intervention is also necessary after mappings have been generated, since several alternative ways of mapping the source into the target may exist. Mapping systems usually produce all alternatives, and offer the user the possibility of inspecting the result of each of them in order to select the preferred one. Based on such an architecture, Figure 1 illustrates the overall mapping process and highlights the phases in which user intervention is required.

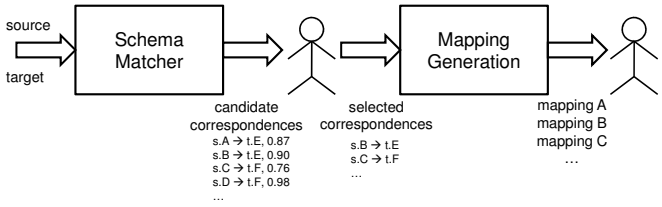

Figure 1: Coupling schema matching and mapping discovery

In this paper, we introduce Spicy, a research project at Università della Basilicata. The main intuition behind our approach is that in many cases the mapping generation process can be automated to a larger extent by introducing a third step, which we call *mapping verification*. More specifically, we assume that instances of the target data source are available, and not only its schema. This is typical in many settings, like, for example, Web data sources. In this case, whenever we select a set of candidate correspondences produced in the schema matching phase, we may think of checking the corresponding source-to-target transformation – i.e., the executable query induced by such correspondences – by running the query on (a subset of) the source, and comparing the result to the available target instance. Based on such comparison, we may on one side identify incorrect transformations due to wrong correspondences produced during the schema matching phase, and on the other side rank the remaining candidates to suggest to users the ones that more likely represent correct translations of the source.

The main contributions brought by the project can be summarized as follows:

(a) Spicy proposes an original architecture to integrate schema matching and mapping generation; so far, these two problems have been studied essentially as independent steps of the integration process;

(b) Spicy represents one of the first proposals towards the definition of a notion of *mapping quality* and the automation of *mapping verification*; we believe that such notions are crucial in order to improve the quality of current integration systems; the paper introduces an algorithm that combines schema matching, mapping generation and mapping verification in order to achieve good scalability and high matching quality;

(c) in view of this, Spicy introduces an original approach, called *structural analysis*, to the comparison of data sources; structural analysis uses electrical circuits to compare the topology and the information content of tree-like structures in order to have a quick measure of their similarity; in the paper, we show that structural analysis represents a very powerful tool for mapping verification;

(d) finally, the system architecture is generic and modular by nature; it can handle both flat (i.e., relational) sources, and nested ones, as XML repositories or OWL ontologies; also, it is designed to work in conjunction with any existing schema matching and mapping generation system, thus allowing to leverage the vast body of research in this field.

We believe the issue of coupling schema matchers with mapping generation systems, and the related issue of verifying mapping quality, represent relevant research problems. To the best of our knowledge, Spicy represents the first proposal towards the automated verification of schema mappings. So far, the only step in the direction of verifying mappings produced by a mapping generation system has been presented in [8], under the form of an ad hoc tool that serves as a debugger for the mapping generation process. The tool allows users to trace and inspect mappings step by step during their generation, similarly to source code debuggers. Note that our approach is significantly different from the one pursued in [8], since we aim at reducing human intervention, while human users play a major role in the debugging process.

The paper is organized as follows. Section 2 gives an overview of our approach and introduces the mapping verification algorithm by means of examples. Preliminaries are in Section 3. Structural analysis is introduced in Section 4, and the mapping algorithm in Section 5. Experimental results are discussed in Section 6. A discussion of related works is in Section 7.

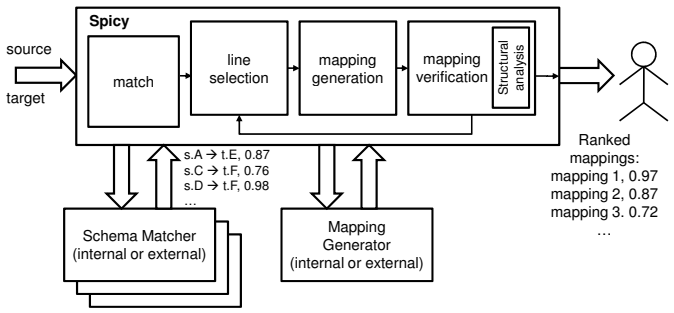

Figure 2: Architecture of Spicy

2. OVERVIEW

The mapping problem we want to solve can be informally described as follows: given (a portion of) a repository T, called the *target*, and a repository S, called the *source*, find a transformation query – i.e., an executable view – that can be used to map instances of S into instances of T. In the process, we aim at minimizing the amount of human intervention. The overall system architecture is shown in Figure 2. In order to solve the problem, we assume the availability of one or more schema matchers, and of a mapping generation module. The main advancement of Spicy with respect to the simple pipelining described in Figure 1 is that these modules are decoupled by a number of intermediate additional modules that coordinate the mapping generation process in order to select the best results.

Given the variety of schema matching systems available, the system architecture has been designed in such a way that any of those can be easily integrated. We have tested the

system using both a *schema-based* and an *instance-based* matcher [29]. As an example of a schema-based tool, the system may be integrated with COMA++ [11, 4]. As there were no instance-based tools available to us at the time of writing, we developed an internal instance-based matcher. Spicy's attribute matcher allows for two different matching strategies: on the one side, it may compare attributes in a rather standard way using features extracted by a sample of their values; on the other side, it may adopt structural analysis, as described below. Since the matching module is not the primary subject of this paper, we will not elaborate further on this issue.

Several points are worth mentioning with respect to the matching phase. First, let us only note that, as it is common in the literature,[1] in this paper we shall mainly concentrate on 1:1 attribute correspondences. However, the Spicy schema matching module also handles a common class of n:1 element correspondences. Note also that, for the sake of simplicity, we do not consider more expressive classes of correspondences, like for example *contextual matches* [6], in which the same source attribute must be matched to different target attributes in different contexts. However, our setting can be extended to handle these cases. Finally, our architecture naturally lends itself to the integration of different schema matchers. However, integrating the outputs of different matchers goes beyond the scope of this paper and is left to future investigations.

The system also assumes the availability of a mapping generation module. The mapping generation module takes as input a set of correspondences, and generates a number of mappings under the form of *source to target dependencies (TGDs)* [28]. These mappings can then be assembled into complex *transformation queries*. A *transformation query* is the union of several mappings; it corresponds to an executable query that can be run over a source instance to obtain a target instance. Examples of transformations are provided below. To derive mappings in Spicy, we have implemented a version of Clio's mapping algorithm [28]. However, other mapping generation tools, like, for example, HePToX [7], could be easily integrated into the system.

The first step of our mapping discovery procedure consists of running the schema matcher to produce a number of candidate element correspondences. Each correspondence is usually labeled with a similarity measure, i.e., a level of confidence. Note that, in most cases, the schema matching system will not be able to produce a single correspondence for each target attribute. It will rather produce a number of compatible source attributes for each target attribute, with different degrees of similarity. Since in our setting each target attribute must be matched to a single source attribute, we need to consider these as alternative lines to be given as input to the mapping generation module.

Example 1: *Consider for example the data sources in Figure 3. As it is common in data integration systems, data structures are represented in the system using an abstract, graph-based model. It can be seen that the data model is essentially a tree-based representation of a nested-relational model, in which set and tuple nodes alternate. Instances are trees as well; while leaf nodes have atomic values – strings, integers, dates etc. – values for intermediate nodes are oids. Foreign key constraints are drawn as dashed edges going from the foreign key to the corresponding primary key.*

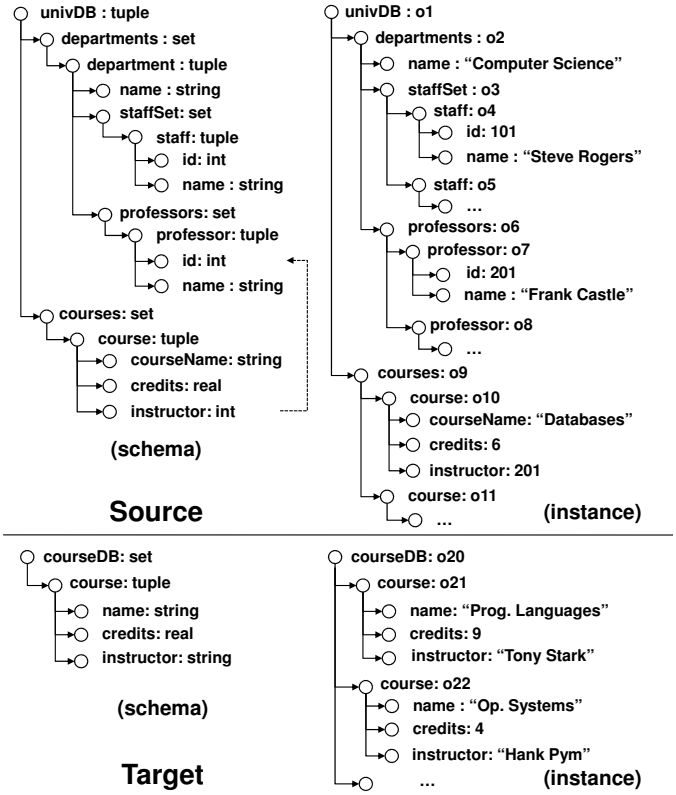

Figure 3: A sample mapping task

Assume that, when run on the two data sources in our example, the selected schema matcher suggests the following correspondences:

courseDB.course.name → univDB.courses.course.courseName, 0.95
courseDB.course.credits → univDB.courses.course.credits, 0.92
courseDB.course.instructor → univDB....professor.name, 0.87
courseDB.course.instructor → univDB ...staff.name, 0.90

It can be seen how we are deliberately assuming that the schema matcher considers staff names to be more similar to course instructors' names than professors' names. It is also apparent that such similarities cannot be fed to the mapping generation algorithm as they are; since attribute course.instructor *has been matched to two different source attributes, the system must choose between two different consistent line sets: in both sets course names and credits are correctly mapped to their counterparts; then, in set* (a) *instructors' names are mapped to professors' names, while in set* (b) *to staff names.*

If we were to select the preferred lines exclusively based on similarity values, we would choose the second candidate set. However, let us reason for a moment on the two transformations that would be produced. How these transformations are generated is discussed in major details in Section 3.2.

Set (a) *yields the following transformation (we adopt the syntax used in [28]):*

[1] In fact, essentially all systems in the literature with a few exceptions [10, 31] are restricted to 1:1 correspondences.

```
for     d in univDB.departments, p in d.professors,
        c in univDB.courses
where   c.course.instructor = p.professor.id
exists  c' in courseDB
where   c'.course.name = c.course.courseName and
        c'.course.credits = c.course.credits and
        c'.professor.name = c'.course.instructor
UNION
for     d in univDB.departments, p in d.professors,
exists  c' in courseDB
where   p.professor.name = c'.course.instructor
```

The transformation is the union of two TGDs. The first one states that, in order to obtain an instance of the target, we need to join courses and professors in the source, and then produce a tuple for each course name, credits and professor name. The second mapping, on the contrary, maps professor names. However, it only contributes to the result for those professors that do not have courses and therefore do not participate to the join above. If, how it is reasonable, we assume that the vast majority of professors have courses, very few tuples will be generated by the second TGD. Based on these observations, we may say that the translated target instance – which we report in tabular form for space reasons – would look as follows:

name	credits	instructor
Databases	6	Frank Castle
Networks	3	Scott Summers
...	...	...

Set (b)*, on the contrary, yields a quite different transformation:*

```
for     c in univDB.courses
exists  c' in courseDB
where   c'.course.name = c.course.courseName and
        c'.course.credits = c.course.credits
UNION
for     d in univDB.departments, s in d.staffSet
exists  c' in courseDB
where   c'.course.instructor = s.staff.name
```

In this case, both mappings contribute to the result, because nobody in the staff teaches courses. The resulting instance is quite different from the previous one, since the two mappings generate a number of null values:

name	credits	instructor
Databases	6	NULL
Networks	3	NULL
NULL	NULL	Frank Castle
NULL	NULL	Scott Summers
...	...	...

If we compare this second instance to the target instance shown in Figure 3, we may see how information "flows" quite differently through them. This might allow us to correctly infer that set (a) *more likely represents the correct choice.*

Although this example has been kept very small and rather simple for clarity's sake, we may summarize by saying that schema matchers typically output correspondences that may be assembled in different ways. This leads to different candidate transformations. We advocate that, while an a priori selection of the right correspondences is often very difficult and typically leads to poor results, an a posteriori comparison of the instances produced by the various transformations to the original target instance may give very useful insights on the quality of these transformations, and therefore on the correctness of the associated lines.

Note that a similar procedure may also help to rank transformations when the right correspondences are known, as shown in the following example.

Example 2: *This example is a variant of the running example used in [28]. Suppose both the source and the target are relational databases, as shown in Figure 4. We assume that correspondences have been identified without ambiguity. Still, the mapping generation algorithm will suggest two different transformations. This is a consequence of the multiple join paths that exist in the source among companies and funds: to each fund we may attach both the company to which the fund was granted and the company that served as a sponsor. As a consequence, the mapping generation algorithm cannot univocally cover correspondences for company names and budgets, and two different mappings are generated, as follows (variables corresponding to aliases have been emphasized):*

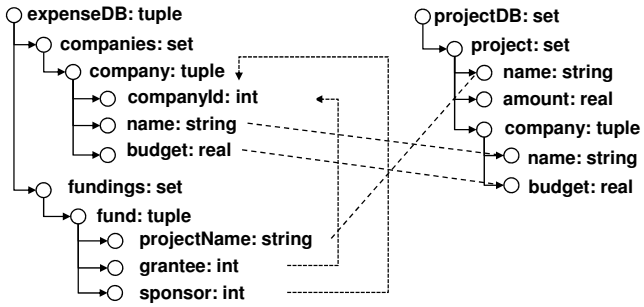

Figure 4: A mapping task with alias

```
for     f in expenseDB.fundings,
        G in expenseDB.companies,
        S in expenseDB.companies
where   f.fund.grantee = G.company.companyId and
        f.fund.sponsor = S.company.companyId
exists  p in projectDB
where   f.fund.projectName = p.project.name and
        G.company.name = p.project.company.name and
        G.company.budget = p.project.company.budget
```

```
for     f in expenseDB.fundings,
        G in expenseDB.companies,
        S in expenseDB.companies
where   f.fund.grantee = G.company.companyId and
        f.fund.sponsor = S.company.companyId
exists  p in projectDB
where   f.fund.projectName = p.project.name and
        S.company.name = p.project.company.name and
        S.company.budget = p.project.company.budget
```

In the first one, the company associated with the project in the target is the grantee ($\mathbf{G}$*), in the second case is the sponsor (*$\mathbf{S}$*). In general, it is not possible to choose one of these mappings without resorting to explicit user feedback [32]. However, also in this case, by looking at the original target instance, we may have some hints. More specifically, companies in the source are divided in two categories: companies that sponsor projects, and companies that do not. It is conceivable that the first ones have considerably higher budgets that the second ones. As a consequence, the first mapping above will produce a translated instance in which*

budgets have lower values on average than the second mapping. Again, by comparing the two instances to the original target instance we may use this knowledge to choose the preferred mapping.

These examples show that a post-translation check of translated instances against the original target instance may help to rank them and suggest the preferred ones. It is still open how such comparison must be conducted. In the following section we introduce a novel strategy to compare trees that serves this purpose.

2.1 Overview of Structural Analysis

Structural analysis is based on the adoption of a suitable model to analyze and compare data structures. Suppose we are given a portion s of a repository S and a portion t of a repository T; in order to compare them, we might proceed as follows: (i) analyze s using the selected model in order to derive a number of descriptive features; (ii) do the same for t; (iii) compare the features computed by the model and return an index of similarity that is as high as close are the selected features.

To do this, we have selected a model that we believe meets two critical requirements – simplicity and good level of abstraction: the model of electrical circuits.[2] The main intuition behind this choice is that a data structure can be seen as a "generator of information"; in essence, we may imagine that its elements – for example, its atomic fields like strings or numbers – tend to produce a flow of information in the repository – i.e., they generate an internal *stress*; each element, in turn, has some *consistency* due to its nature – for example, numbers may be considered of different consistency with respect to strings or dates; from the interaction between stress and consistencies we may predict the flow of information. Intuitively, two instances will be considered to be similar when information flows similarly through them; in this way, circuits allow us to check the validity of mappings as discussed in Examples 1 and 2.

Electrical circuits exhibit nice and elegant properties that make them a very effective means to study similarities about data structures. In the following sections, we formalize a transformation function, that, when applied to a tree-like portion of a data source, generates an electrical circuit that can be used for the purpose of comparison. For now let us mention some features of this transformation.

First, the resulting circuit has a tree-like topology that is isomorphic to that of the original tree; also, the circuit embodies instance-based features; in fact, values for resistors and voltage generators associated with attributes are derived from numerical features of a sample of instance values; hence, it is an ideal means to model the informative content of the data structure.

Second, circuits are an excellent summarization technique for the purpose of verifying mappings. Recall that our goal is to evaluate the quality of a mapping by checking the similarity of the instance obtained by executing the mapping with respect to the instance originally provided with the target data source. To do this, in principle, we might adopt a rather straightforward attribute-to-attribute comparison approach, as follows: (i) after the translated instance has been generated, we might consider each attribute A_i in the target separately; (ii) take a sample of values for A_i in both the original and the translated instance, and derive a number of numerical features based on this sample; (iii) calculate the overall similarity between the two instances based on the similarity of the various features. However, combining independent similarity measures into a single similarity value is known to be a hard problem [18]. Circuits provide a nice and elegant solution to this problem. They naturally lend themselves to a black-box analysis to characterize the behavior of the two trees, through which we can collect a small number of descriptive parameters – output current, average current, internal voltages – that can be effectively used to measure the similarity of the original structure to other structures. Our experiments show that circuits perform better than attribute-to-attribute feature comparisons.

Finally, the comparison technique based on circuits is fully automatic and does not require any additional inputs; it is also quite efficient, since, for the kind of stationary, continuous current circuits used in the system, solving the circuit amounts to solve a system of linear equations of the form $Ax = b$.

2.2 The Mapping Search Algorithm

We now give an intuition of the mapping discovery and verification process pictured in Figure 2. A key observation, here, is that the mapping generation process can be quite demanding from the computational viewpoint. As a consequence, it is mandatory to design the search algorithm in such a way to achieve a good trade-off between precision and scalability. Recall that our starting point is a number of candidate correspondences generated by the schema matching module. Whenever multiple source attributes are matched – possibly with different degrees of similarity – to the same target attribute, we need to generate alternative transformations and then rank them. The higher is the number of correspondences produced by the schema matching step, the more mappings will be generated and verified.

In view of this, our algorithm is based on a feedback loop in which we initially fix a similarity threshold; at each step: (i) we consider only candidate correspondences with similarity above the threshold; (ii) we combine these correspondences into consistent sets, and run the mapping generation algorithm to generate the corresponding transformations; (iii) we use each transformation to translate (a sample of) the source instance into an instance of the target, and then use structural analysis to compare the obtained instances to the original target instance; (iv) we rank transformations – and therefore correspondence sets – according to such similarity measure. If no satisfactory mappings are produced at a given iteration, we progressively lower the threshold and analyze more alternative mappings. In essence, using this process, users are not required to manually inspect and select correspondences before mappings are generated; also, when the process stops, the system will present to them a ranked list of source-to-target transformations, suggesting which ones are believed to better reproduce the target.

[2] In fact, the name of the project is inspired by SPICE, the well-known circuit emulator.

A key idea in this process is that outputs of the schema matching module are not considered as a definitive proof that a source and target attribute are semantically equivalent, but rather as a measure of compatibility between them. Such compatibility is then checked in subsequent steps. Our experimental results show that our verification algorithm achieves very good precision and allows to handle many errors due to incorrect matches.

The following sections introduce with greater detail the models and algorithms used in the system.

3. PRELIMINARIES

This section introduces some preliminary notions that will be used in the rest of the paper.

3.1 Data Model

As it is common, data sources are represented in the system according to an abstract model, with the advantage of reducing different concrete data models (relational, XML, OWL etc.) to a uniform representation. In the following we briefly formalize such data model.

We fix a number of *base data types*, t_i – e.g., string, integer, date etc. – each with its domain of values, $dom(t_i)$, a set of *labels*, $A_0, A_1 \ldots$, and a set of special values, called oids. A type is either a base type or a set or tuple complex type. A *set type* has the form $\mathsf{set}(A : \tau)$, where A is a label and τ is a type. A *set node* in a schema is a labeled set type, of the form $A : set(\tau)$, with a child node $A : \tau$. A *tuple type* has the form $\mathsf{tuple}(A_0 : \tau_0, A_1 : \tau_1, \ldots, A_n : \tau_n)$, where each A_i is a label and each τ_i is a type. A *tuple node* is a labeled tuple type with a child node $A_i : \tau_i$, for $i = 0, 1 \ldots, n$. Schemas are trees of typed nodes. More specifically, a *schema* is a named type $A : \tau$. Constraints may be imposed over a schema. A *constraint* is either a key or a foreign key constraint.

Instances of types are defined as usual. An instance of a schema tree is a tree of instance nodes. As in [28], an instance node for a schema node of type τ, has a distinct oid value, plus a number of children, according to the respective type. The sample data sources shown in Figure 3 are compliant with this model. It can be seen as a relational database is easily represented as a root tuple node, plus a number of children set nodes, one for each relation. We also represent nested XML structures. In this paper, we restrict our attention to the case in which XML schemas are single-rooted and contain only set and tuple/sequence types.

3.2 Mapping Discovery Algorithm

The mapping discovery employed in the system is the one developed in Clio. In the following, we briefly review the main steps of the algorithm [28]. Given a source and a target schema, the algorithm produces a number of executable queries. Each query is the union of several mappings that represent source-to-target tuple generating dependencies (TGDs). In order to identify such dependencies, several preliminary steps are necessary.

As a first step, we need to identify the *logical relations* both in the source and target schemas. Logical relations are maximal tableaux [2]. In order to generate them, the algorithm finds the so called *primary paths* in each schema. These are essentially linear tableaux, and are obtained by the enumeration of all paths from the root to any intermediate node of set type in such a schema. With respect to the schemas in Figure 3, the primary paths generated for the source and target schemas respectively, are the following (with an abuse of notation we use "select *" to refer to all attributes that are directly reachable from a set node in the tableaux):

```
select * from d in univDB.departments
select * from d in univDB.departments, s in d.department.staffSet
select * from d in univDB.departments, p in d.department.professors
select * from c in univDB.courses

select * from p in courseDB
```

Logical relations are obtained by chasing constraints against primary paths. In our example, a single constraint must be considered, from univDB.courses.course.instructor to univDB.departments.department.professors.professor.id. Thus, we obtain the following logical relations:

```
select * from d in univDB.departments
select * from d in univDB.departments, s in d.department.staffSet
select * from d in univDB.departments, p in d.department.professors
select * from c in univDB.courses,
        d in univDB.departments, p in d.department.professors
where   c.course.instructor = p.professor.id

select * from p in courseDB
```

During the third step, inter-schema correspondences are processed to yield a set of mappings. To generate mappings, all possible pairs made of a source logical relation and a target logical relation are considered; a pair generates a mapping if it covers one or more value correspondences. As an example, assume the following correspondences have been provided to the algorithm for the mapping task in Figure 3 (similarities are omitted since they are not strictly relevant to the mapping generation algorithm):

courseDB.course.name → univDB.courses.course.courseName
courseDB.course.credits → univDB.courses.course.credits
courseDB.course.instructor → univDB....professor.name

The algorithms will generate the following TGDs:

```
for     d in univDB.departments, p in d.professors,
        c in univDB.courses
where   c.course.instructor = p.professor.id
exists  c' in courseDB
where   c'.course.name = c.course.courseName and
        c'.course.credits = c.course.credits and
        c'.professor.name = c'.course.instructor

for     d in univDB.departments, p in d.professors,
exists  c' in courseDB
where   p.professor.name = c'.course.instructor
```

Such TGDs represent different ways to cover the correspondences and generate tuples in the target. The final transformation query is obtained by taking the union of the two. Note that, in general, some of the mappings generated by the algorithm are redundant, since they are either *subsumed* or *implied* [16] by other mappings. Such mappings should be discarded. Once transformations have been generated as unions of TGDs, it is then rather straightforward to translate this abstract syntax into a concrete one, for example XQuery.

Note, however, that special care must be taken to avoid the generation of duplicates in the target and to correctly group

translated values. Several strategies have been proposed to this end (see, for example, [16]). In our system, besides the ordinary translation to XQuery, we have decided to implement an internal execution engine to process transformations. Similarly to Clio [20], the execution engine employs a deep union operator to prevent the generation of duplicates.

4. STRUCTURAL ANALYSIS

To compare repository portions we transform their trees into circuits, solve the circuits, and then compare features of the two circuits (e.g., output currents). In the following, we first introduce the circuit mapping function, and then the actual compare procedure.

4.1 Electrical Circuits

We consider electrical circuits [9] made of elements connected to nodes. In our case, elements may be voltage generators and resistors. In such circuits, voltage generators cause a flow of current through resistors. A voltage generator imposes a constant difference V_k of electric potential between two nodes. A resistor of value r causes a drop of potential V at its nodes due to the current I that flows through it; current and voltage are related by *Ohm's law* $V = rI$. A circuit usually has two distinctive nodes: (i) a *ground node*, whose potential is conventionally 0; voltages at other nodes are measured as differences of potential with respect to the ground; (ii) an *output node*, i.e., a node at which output voltage and current are measured. We define the *output current* of the circuit as the current flowing through the output node when an *external resistor* of fixed value is added from output to ground. Solving the circuits amounts to calculating voltages at the nodes and currents through elements. This corresponds to solving a system of linear equations based on Kirchhoff's law of size $n \times n$, where n is the number of nodes in the circuit.

4.2 Sampling

In order to build the circuit associated with a data source, we need to sample instances. For each leaf A in a tree we select a random sample of instances, $sample(A)$ of size K (or less) from the repository. Two attributes are considered to match when they represent the same concept, and their values are coded using the same format and conventions. As a consequence, we are interested in studying features such as the length of values in the sample and the distribution of characters. For each attribute A we build a number of distributions. The *distributions of lengths* $\mathcal{D}_L(sample(A))$ contains the frequencies of lengths in the sample, measured in bytes. Fixed an ordered collection of character categories $\mathcal{C} = \{c_0, c_1, \ldots c_{n-1}\}$ (i.e., letters, capitalized letters, digits, special characters, at-signs, slashes etc.), we also compute the *distribution of character categories*, $\mathcal{D}_\mathcal{C}(sample(A)) = \{f_{c_0}, f_{c_1}, \ldots f_{c_{n-1}}\}$, where f_{c_i} is the frequency of characters belonging to category c_i in the sample.

Given a distribution of frequencies $\mathcal{D} = \{f_0, f_1, \ldots, f_{n-1}\}$ such that $\sum_{i=0}^{n-1} f_i = 1$, we measure several statistical parameters. Besides the usual ones (mean value, standard deviation, mode), we also consider the *Simpson concentration index* [27] of the distribution, defined as $IC(\mathcal{D}) = \sum_{i=0}^{n-1} f_i^2$; intuitively, this index gives a measure of the entropy of values in the distribution, in the spirit of information theory: the higher is the entropy in the sample, the lower is the concentration index.

Finally, we define the *density* of a sample of values of attribute A, $density(sample(A))$ as n/K, where K is the size of $sample(A)$, and n is the number of non-null values in $sample(A)$. This is a measure of the number of null values generated by a transformation, as discussed in Example 1.

4.3 Circuit Generation Function

We introduce a recursive function $circ(t)$, that, given a tree t generates a circuit. In order to do this, we need to formalize the output of $circ()$ on an atomic attribute, i.e., a leaf in the tree. There are several ways to map a sampled attribute to a circuit; intuitively, this depends on the features one decides to embed into the circuit, and on the topology of circuit elements that represent them. In the following, we describe the strategy that has provided the best results in our experiments. Given an attribute A in a schema tree, annotated with a sample of values, $sample(A)$, with length and character distributions $\mathcal{D}_L$ and $\mathcal{D}_\mathcal{C}$ respectively, we define the following features:

- *density index* (**ID**): $ID(A) = density(sample(A))$;
- *consistency* (**C**): $C(A) = 2^0 \times f_{c_0} + 2^1 \times f_{c_1} + \ldots 2^{n-1} \times f_{c_{n-1}}$, where $c_0, c_1, \ldots c_{n-1}$ are character categories, and f_{c_0}, $f_{c_1}, \ldots f_{c_{n-1}}$ the respective frequencies in $\mathcal{D}_\mathcal{C}(sample(A))$; for the sake of convenience we will refer to the polynomial function above simply as $C(A) = pol(\mathcal{D}_\mathcal{C}(sample(A)))$;
- *stress* (**S**): $S(A) = mean(\mathcal{D}_L(sample(A))$, i.e., the mean value of the distribution of lengths in the sample;
- *conc. index 1* (**IC1**): $IC1(A) = IC(\mathcal{D}_\mathcal{C}(sample(A))$, i.e., the index of concentration of the distribution of character categories;
- *conc. index 2* (**IC2**): $IC2(A) = IC(\mathcal{D}_\mathcal{L}(sample(A))$, i.e., the index of concentration of the distribution of lengths.

$circ(A)$ is assembled by assigning these features to a number of resistor and voltage generators, as described in Figure 5.

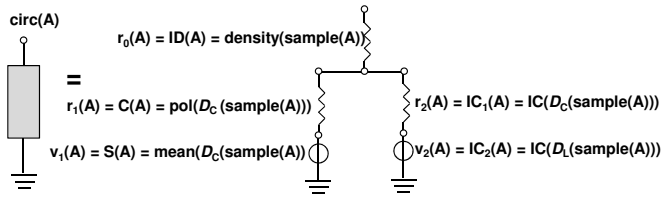

Figure 5: Electrical circuit for an atomic attribute

The circuit for a tree is easily constructed starting from building blocks corresponding to atomic attributes; more specifically, for each intermediate node n in a schema tree t, we define a resistance value, $r(n)$. Such value cannot be based on instances, since intermediate nodes do not have a sample of instances, but rather on the topology of the tree. More specifically, we define $r(n) = k \times level(n)$, where k is a constant multiplicative factor, and $level(n)$ is the *level*

of n in t, defined as follows: (i) leaves have level 0; (ii) an intermediate node with children $n_0, n_1, \ldots n_k$ has level $max(level(n_0), level(n_1), \ldots level(n_k)) + 1$; $nodes(t)$ will denote the set of nodes in tree t.

We can now give a complete definition of the *circuit mapping function*, $circ(t)$ over a tree t. For a leaf node A, $circ(A)$ is as defined above. For a tree t rooted at node n with children $n_0, n_1, \ldots n_k$, $circ(t)$ is the circuit obtained by connecting in parallel $circ(n_0), circ(n_1), \ldots circ(n_k)$ between ground and an intermediate circuit node n_{top}, and then adding a resistor of value $r(n)$ from node n_{top} to output. Examples of such transformation are given in Figure 6. Note how the resulting circuits are essentially isomorphic to the original trees.

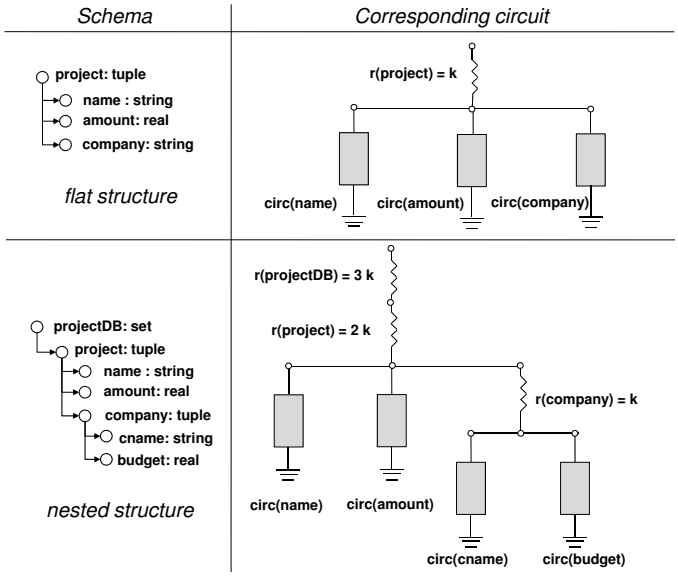

Figure 6: Examples of Circuits

Note that, coherently with the *opaque* [21] nature of our approach, labels are not taken into account by the circuit mapping function, and we are treating values essentially as uninterpreted strings. Also, in this paper we concentrate on ordinary alphanumeric data: the features discussed above reflect this choice. However, the circuit model is sufficiently flexible to allow the treatment of special data, like large texts or multimedia, as well; for these data different features must be adopted [15].

4.4 Compare Procedure

Similarly to [12], our compare module adopts a modular library of comparators, that can be mixed and matched in various ways. Given two trees t_1 and t_2, we compute a measure of their similarity, as follows: (i) map t_1 and t_2 to the corresponding circuits, $circ(t_1), circ(t_2)$; (ii) solve the two circuits to determine currents and voltages; (iii) choose a number of descriptive features of the corresponding circuits, $f_0, f_1, \ldots f_i$; we introduce the notion of a *comparator* for feature f_i as a module that computes the index of similarity Δ_i between the two structures with respect to feature f_i, as follows $\Delta_i = abs(f_i(circ(t_1)) - f_i(circ(t_2)))/f_i(circ(t_1))$; (iv) finally, we compute the overall similarity of the two trees based on $\Delta_0, \Delta_1, \ldots \Delta_i$.

A very delicate problem in this setting is represented by assembling different similarity measures into an overall quality estimate; to this end, the system provides three alternative strategies to compute an overall similarity measure based on a collection of feature similarity values like $\Delta_0, \Delta_1, \ldots \Delta_i$, namely: standard average (A), i.e., arithmetic mean, of similarity values; harmonic mean (HM) of similarity values; euclidean distance (D); in this case, candidates are considered as points in an n-dimensional space, whose coordinates correspond to $\Delta_0, \Delta_1, \ldots \Delta_i$; euclidean distances give a measure of how far away a point is from the origin or from another point; in this case two points are considered to be the more similar the less is their distance.

Using such a setting, we may compare circuits – and therefore trees – using different collections of comparators and aggregation strategies. To give an example, suppose we decide to use output current alone as a comparison feature. In this case, we would have a single comparator, and similarity would be equal to the percentual difference between the output currents measured in the two circuits. If we decide to employ both output current and average current, than we have two different comparators, and we need to choose an aggregation strategy – say, harmonic means; the overall compare will be the harmonic means of $\Delta_{outCurr}$ and $\Delta_{avgCurr}$. Similarly for other features. In Section 6 we discuss how the selection of comparators affect precision.

5. MAPPING SEARCH ALGORITHM

We can now describe the mapping discovery and verification process pictured in Figure 2.

A *mapping task* for SPICY is a pair of data sources, $< S, T >$, where S is the source and T the target. As a first step, the source and target are submitted to the schema matching module, and a list of candidate correspondences is generated. This step is performed only once for each mapping task. The line selection module of SPICY stores the result of the match step, and generates inputs to the subsequent phases. Typically, it will only consider a subset of correspondences, those with a confidence level above a fixed threshold, th.

As discussed in Section 2, it is possible that multiple source attributes are matched – possibly with different degrees of similarity – to the same target attribute. We call *unambiguous* a collection of correspondences if each attribute – both from the source and the target – appears in at most one of the correspondences. Correspondences generated by the matching module contain in general ambiguities, and therefore need to be partitioned into a number of alternative unambiguous sets. Each unambiguous set of correspondences will then be fed to the mapping generation module, and will produce a transformation query. These transformations need to be ranked by running structural analysis.

Before turning to the description of the actual algorithm, we need to make some preliminary observations. A key point, here, is that the mapping generation process can be quite demanding from the computational viewpoint. As a consequence, it is mandatory to design the algorithm in such a way to achieve a good trade-off between precision and scalability. It can be easily seen that the higher is the number of correspondences that are considered, the higher is the prob-

ability of finding ambiguities, and therefore the number of instances of the mapping generation process to execute. It is therefore very important to choose the confidence threshold, th, in such a way to be selective enough and discard poor mappings, without excluding promising ones. High values of the threshold generate a few candidate mappings, potentially none, and may reduce precision. Low values increase the number of candidates, and may impose a significant overhead in terms of computing times. In fact, experiments confirm that precision tends to increase as the threshold decreases. However, this improvement in precision has a cost in terms of computing times: the number of candidate mappings tends to increase very rapidly, and computation times with them.

Based on these results, it can be seen that it is very difficult to statically fix an optimal value for the compatibility threshold th. In view of this, we adopt a *dynamic threshold*, initially very high in order to consider only a few candidate correspondences; then, if no satisfactory mapping is produced, we progressively lower the threshold and analyze more alternative mappings. In essence, the mapping algorithm starts with a very high value for the compatibility threshold (0.99). If no mapping of acceptable quality is produced, the algorithm backtracks, it lowers the threshold (of 0.01), and iterates. By lowering the value of th, more correspondences are taken into account, and therefore more potential mappings. This process proceeds until either a mapping is produced, or the threshold falls below a stop value, th_{stop} (0.6).

Given a mapping task $< S, T >$, the algorithm can be sketched as follows:

1. run the external schema matcher module to generate candidate correspondences, $match(S, T)$;
2. fix a step, *step*, and a stop value th_{stop} for the compatibility threshold; initialize $th = 1 - step$; fix a minimum value for mapping quality $minQ$;
3. consider all correspondences in $match(S, T)$ with confidence above th; assemble them into maximal unambiguous subsets;
4. feed each subset of lines $\mathcal{S}_i$ to the mapping generation module, and generate the corresponding transformations (one or more, depending on the presence of aliases in the data sources); process each transformation using the internal execution engine to translate a sample of the source instance into an instance of the target;
5. sample each translated instance and use structural analysis as discussed in Section 4 to compare it to the original target instance; the quality of the associated transformation is given by the compare value;
6. if no transformation with quality above $minQ$ is produced, then $th = th - step$, and iterate step 3; otherwise output all transformations with quality above $minQ$, ranked according to their estimated quality.

Note that at each step a number of candidates that have already been processed in previous steps are re-generated. An aggressive caching strategy has been implemented into the system in order to improve efficiency. More specifically, we cache logical relations, mappings and circuits in order to avoid repeated computations along the path. This is quite effective in reducing the time cost but makes the algorithm quite demanding in terms of memory.

6. EXPERIMENTAL RESULTS

The system described in the paper has been implemented in a working prototype. The prototype was developed in Java. We have used the prototype to run a number of experiments. Table 1 summarizes the list of experiments that were used to test the system. We analyzed 12 mapping tasks, based on different data sources, both relational and XML. Besides well known data sources like DBLP, Mondial [3], and Amalgam [4], we tested several real–life databases serving the information system of our School of Computer Science (CS-IS, LbDb, Moodle), and some synthetic datasets.

Source	S. Type	Target	Targ Type	S. size (nodes)	T. size (nodes)
LbDb	rel	Comp. Science IS	rel	54	6
Moodle	rel	LbDb	rel	596	6
Comp. Science IS	rel	LbDb	rel	276	4
Amalgam1	rel	Amalgam2	rel	18	6
DBLP	xml	Amalgam1	rel	89	5
DBLP	xml	Amalgam2	rel	89	5
Mondial	xml	CIA Factbook	xml	222	14
Mondial	xml	Global Statistics	xml	222	4
Mondial	xml	Mondial Flat Europe	xml	222	9
Census1	xml	Census2	xml	18	6
UniversityDB	rel	Personnel	rel	28	5
StatDB	xml	ExpenseDB	rel	25	13

Table 1: Summary of Experiments

Experiments were run on an Intel core-duo processor machine, with 2 GB of RAM. Mapping tasks were designed in such a way that the source was known to contain a mapping for the entire target. For each target, we identified the correct set of correspondences that would generate such mapping. These correspondences were called the *ideal match* $\mathcal{M}_{id}$; then, we ran the system, and considered as output a single transformation, T_{best}, the one with the highest similarity score. We considered the value correspondences from which T_{best} was generated, called $\mathcal{M}_{T_{best}}$. We measured quality in terms of precision and recall of $\mathcal{M}_{T_{best}}$ with respect to $\mathcal{M}_{id}$. Note however that, in this section we report precision results only. This is due to the fact that in all cases the system returned a number of correspondences that was equal to the size of the target (combinations with less correspondences were discarded since they lowered the similarity to the target instance due to excessive presence of nulls); as a consequence, precision and recall are both equal to the number of correct correspondences in $\mathcal{M}_{T_{best}}$ over the size of the target.

A key issue to validate our approach was to compare the quality obtained by selecting attribute matches *a posteriori*,

[3] http://www.dbis.informatik.uni-goettingen.de/Mondial/
[4] http://www.cs.toronto.edu/~miller/amalgam/

i.e., after mapping generation and translation, with respect to selecting them *a priori*, i.e., relying only on the output of the attribute matcher. To test this, we have run our experiments with several configurations.

Average Match (AM): the best line set was chosen a priori, as follows: the attribute matcher was run to find candidate correspondences; then, the unambiguous line sets (one or more) with the highest confidence were selected; attribute similarities were aggregated in various ways to give variants of this configuration: **AM-A** (average match computed as average), **AM-HM** (average match computed as harmonic mean), **AM-D** (average match computed as euclidean distance). An alternative to these approaches would be to adopt a *top-K* strategy, as in [17]. These configurations were applied to the various schema matchers employed by the system, i.e., Spicy's instance-based matcher based on attribute features **Spicy-F**, and Spicy's matcher based on structural analysis, **Spicy-SA**.[5]

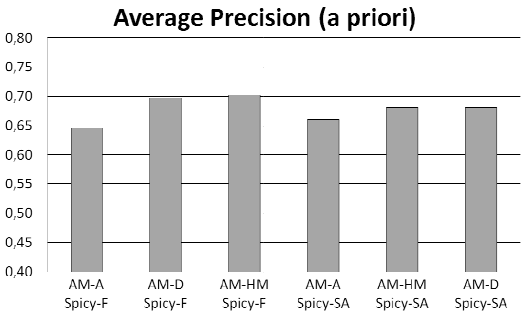

Figure 7: Precision of a priori strategies (average match)

To assess a posteriori strategies, a second, important point, is how structural analysis performs with respect to more traditional comparison procedures, based on attribute features alone. To do this, we have tested two different configurations:

Attribute Features (AF): the best line set was chosen a posteriori, as follows: the mapping find algorithm was run as described in Section 5; however, after candidate transformations were run on the source to obtain a translated instance of the target, their quality was measured by comparing it to the original target instance in a standard fashion, i.e., by comparing each attribute in the translated instance to its counterpart using instance-based features; we adopted the same features that had been used to compare source and target attributes during the match phase (**ID, S, C, IC1, IC2**, as defined in Section 4), and then assembled the similarity measures using average and harmonic means (**AF-A, AF-HM**). Since, as discussed in Section 4, our comparison algorithm has been designed to allow for the combination of different comparators, we tested different combinations of these.

Structural Analysis (SA, LSA): finally, we used structural analysis – i.e., circuits – as described in the previous sections. Among the many different combinations of comparators based on circuit features, we found these two to be the most interesting: **SA**: the current in each branch of one circuit is compared to the current in the corresponding branch of the second circuit; **LSA**: the *global* output current, i.e., the output of the whole circuit was considered, plus the *local* output currents, i.e., output currents in each subcircuit corresponding to an attribute.

[5]We have also run several preliminary experiments using COMA++. We saw that the performance of COMA degrades on opaque schemas or schemas with very different vocabularies. Overall, its performance was in line with those of Spicy's instance-based matchers.

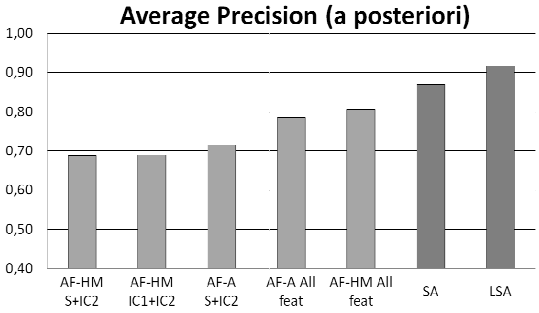

Figure 8: Precision of a posteriori strategies (attribute features and structural analysis)

Average precision over the 12 experiments for a priori strategies is shown in Figure 7, while average precision for a posteriori evaluations is shown in Figure 8. As a first observation, note that, as it was expected, all a priori configurations had mediocre performance – below 70%, while significantly higher values of precision were obtained by some a posteriori configurations. This confirms the intuition of Examples 1 and 2 in Section 2: since the schema matcher does not take into account semantic mappings, as a posteriori verification strategies do, it is frequently mistaken.

A second key observation is that structural analysis had excellent performance: both configurations based on circuits outperformed attribute features, whose best precision was around 80%, while the LSA configuration has precision above 90%, thus confirming the effectiveness of circuits.

More results are shown in Figure 9, in terms of average stop threshold and total execution times. It can be seen that verification strategies based on structural analysis have lower stop thresholds on average; intuitively, this is due to the fact that, given the search algorithm discussed above, they perform deeper searches before finding high quality solutions; this also explains why they achieve higher precision than other strategies. This higher level of accuracy comes at the price of higher execution times. This increase, however, is acceptable, since under the **LSA** configuration the 12 experiments were run in less than 3 minutes.

7. RELATED WORK

Debugging schema mappings is a recent research topic and has been addressed in [8]. There, schema mappings can be traced and inspected, similarly to source code instructions. Their approach gives the user a major role in debugging the schema mappings, thus being significantly different from ours.

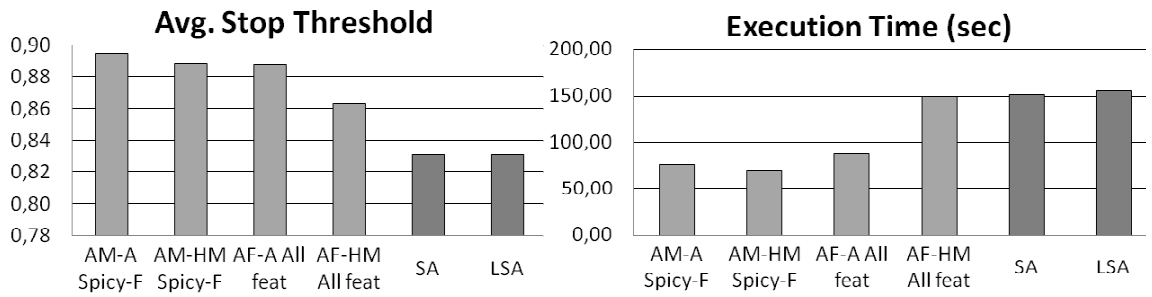

Figure 9: Stop thresholds and execution times

In [6], contextual schema matching is introduced, to denote correspondences annotated with a predicate saying when the match is valid. These conditions are then translated into actual views, and the mapping generation is done by extending the Clio algorithm. We currently do not deal with contextual matches, but our setting can be extended to incorporate such kind of matches.

The fact that schema matching tools may return uncertain results has inspired an active body of research [17, 19]. In [17], it is highlighted that a schema matcher often tries to derive a single best set of correspondences, whereas in most cases the discarded correspondences may convey useful information. An heuristic based on confidence values is used to verify the top-K mappings yielded by a schema matcher, and refine the matches accordingly.

Our system adopts a Clio-style algorithm for mapping generation. Clio [24, 28] does not offer a module for automatic mapping verification, but supports an interactive mapping refinement process by visualizing mapping examples, i.e. smaller samples of the source instance that has been translated using the current mapping. These may help the user to select among alternative solutions.

Many tools for semi-automatic schema matching have been proposed in the past. For a complete reference, we refer the reader to comprehensive surveys in [29, 11, 30]. In the following we shall briefly review some of these tools. Schema matching systems are usually classified in two main categories: *schema-based* systems use a combination of linguistic and graph-based techniques in order to find similarities in schema labels; *instance-based* ones rely on actual values in instances to derive attribute features and similarities. Although schema-based tools represent the vast majority, instance-based tools perform better than schema-based ones in those contexts in which the databases are essentially *opaque* [21], i.e., labels and/or values are difficult or impossible to interpret.

Similarity Flooding [23] (SF, for brevity) proposes a structural algorithm that can be used to compute similarities between arbitrary data structures, such as schemas, instances or both. Despite the apparent similarity between the terms "flood" and "current flow", the two systems have hardly any points in common. Given two structures to compare – called models – SF runs a fixpoint algorithm over an auxiliary data structure, called a *similarity propagation graph* in which elements of the first and second model are embedded together; similarities are propagated along the edges of such graph according to the intuition that two nodes are similar when their adjacent elements are similar. On the contrary, in Spicy, the two tree structures to compare are kept separate, and no common data structure is employed; moreover, each tree is turned into an isomorphic circuit, and the circuit is solved to calculate currents and voltages; this is based on a very different intuition with respect to SF, namely that currents model the "flow of information" inside the tree; no similarity flows through the circuit. Similarities in Spicy are explicitly computed by selecting a number of features that describe the circuits response and measuring their distance.

COMA++ [12] bases on well-founded software engineering principles to build a fully-fledged set of matching techniques. It introduces the notions of reuse and composition of matchers. Both SF [23] and COMA++ [12] are schema matchers, thus can be used within our matching module.

Among the instance-based mapping tools, we recall [5], [21], [22], and [13]. A more exhaustive survey is beside the scope of this paper and can be found in [11]. DUMAS [5] exploits the presence of duplicates within relations to effectively drive the schema matching process. Despite their actual labels, attributes that are semantically similar are detected and used to output the mapping correspondences. Spicy does not assume the presence of duplicates.

In [21], the authors develop an instance-based approach to matching opaque databases, i.e., database in which labels and/or values are difficult or impossible to interpret. They measure the pair-wise attribute correlations in the tables and use mutual information and entropy to build a dependency graph, which is then explored to find matching node pairs. While such information-theoretic approach has some points in common with our use of entropy, note that Spicy does not exploit mutual entropy to find matches. Both [5, 21] are orthogonal to Spicy, which can be used to verify the outcome of the formers.

SemInt [22] clusters similar attributes by using data patterns and catalog information as inputs to neural network. The entire approach is automated, yet a set of parsers need to be instructed at the beginning of a matching task.

LSD [13] adopts a machine learning approach: a set of learners must be trained by feeding examples of mappings among a smaller set of sources. Then, the accumulated knowledge is reused to automatically derive mappings between other sources in the same domain. The instance-based module in Spicy is based on electrical circuits and does not require ad-

ditional input, nor training in order to verify the mappings.

Finally, the idea of using electricity to address computer science problems has also been exploited in other cases. One example are graphs random walks [14, 26]. In [3], the author builds an Hex-playing machine: Hex is a two-player game that aims at connecting two opposite sides of a rhombic board through continuous black or white cells. In the paper, a two-dimensional electrical charge distribution is associated with any given Hex cell. This machine made decisions based on properties of the corresponding potential field.

8. CONCLUSIONS

In this paper we have introduced a novel architecture that makes a step towards the goal of automating mapping discovery. A key idea in our proposal is that, when lines are suggested by a schema matching module, candidate mappings must in general be verified by a dedicated mapping verification module; we showed that this check gives better precision if performed a posteriori, i.e., after the candidate translation query has been run on the source. In Spicy, mapping verification is based on structural analysis, that has proven to be an effective tool to compare data structures.

We envision several future directions of investigation, among which we mention: testing new features to handle non conventional datatypes with circuits, in the spirit of [15]; considering more complex classes of correspondences, as in [6].

Acknowledgments The authors would like to thank Paolo Papotti who provided many insightful comments on the subject of the paper. Salvatore Raunich was partially supported by the European Social Fund (Fondo Sociale Europeo).

9. REFERENCES

[1] The Ontology Alignment Evaluation Initiative – 2007. http://oaei.ontologymatching.org/2007/.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] V. V. Anshelevich. A Hierarchical Approach to Computer Hex. *Artif. Intell.*, 134(1–2):101–120, 2002.

[4] D. Aumueller, H. Do, Massmann S., and E. Rahm. Schema and Ontology Matching with COMA++. In *Proc. of ACM SIGMOD*, pages 906–908, 2005.

[5] A. Bilke and F. Naumann. Schema Matching using Duplicates. In *Proc. of ICDE*, pages 69–80, 2005.

[6] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting Context into Schema Matching. In *Proc. of VLDB*, pages 307–318, 2006.

[7] A. Bonifati, E. Q. Chang, T. Ho, L. Lakshmanan, and R. Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *Proc. of VLDB*, pages 1267–1270, 2005.

[8] L. Chiticariu and W. C. Tan. Debugging Schema Mappings with Routes. In *Proc. of VLDB*, pages 79–90, 2006.

[9] P. R. Clayton. *Fundamentals of Electric Circuit Analysis*. John Wiley & Sons, 2001.

[10] R. Dhamankar, Y. Lee, A. H. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *Proc. of ACM SIGMOD*, pages 383–394, 2004.

[11] H. H. Do, S. Melnik, and E. Rahm. Comparison of Schema Matching Evaluations. In *Proc. of the GI Workshop on Web and Databases*, pages 221–237, 2002.

[12] H. H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proc. of VLDB*, pages 610–621, 2002.

[13] A. H. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *Proc. of ACM SIGMOD*, pages 509–520, 2001.

[14] P. G. Doyle and J. L. Snell. Random Walks and Electric Networks. In *Proc. of the Mathematical Associations of America*, 1984.

[15] C. Faloutsos. Indexing multimedia databases. In *Proc. of ACM SIGMOD*, page 467, New York, NY, USA, 1995. ACM Press.

[16] A. Fuxman, M. A. Hernández, C. T. Howard, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *Proc. of VLDB*, pages 67–78, 2006.

[17] A. Gal. Managing Uncertainty in Schema Matching with Top-K Schema Mappings. *J. of Data Semantics*, VI:90–114, 2006.

[18] A. Gal. Why is Schema Matching Tough and What We Can Do About It. *Sigmod Record*, 35(4):2–5, 2006.

[19] A. Gal. The Generation Y of XML Schema Matching (Panel Description). In *Proceedings of XML Database Symposium*, pages 137–139, 2007.

[20] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: from Research Prototype to Industrial Tool. In *Proc. of ACM SIGMOD*, pages 805–810, 2005.

[21] J. Kang and J. F. Naughton. On Schema Matching with Opaque Column Names and Data Values. In *Proc. of ACM SIGMOD*, pages 205–216, 2003.

[22] W. S. Li and C. Clifton. SEMINT: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases using Neural Networks. *Data and Know. Eng.*, 33(1):49–84, 2000.

[23] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proc. of ICDE*, pages 117–128, 2002.

[24] R. J. Miller, L. M. Haas, and M. A. Hernandez. Schema Mapping as Query Discovery. In *Proc. of VLDB*, pages 77–99, 2000.

[25] F. Naumann, C.-T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute Classification Using Feature Analysis. In *Proc. of ICDE*, page 271, 2002.

[26] C. R. Palmer and C. Faloutsos. Electricity-Based External Similarity of Categorical Attributes. In *Proc. of PAKDD*, pages 486–500, 2003.

[27] R. Pierce. *An Introduction to Information Theory*. Dover Publications, 1980.

[28] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *Proc. of VLDB*, pages 598–609, 2002.

[29] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB J.*, 10:334–350, 2001.

[30] P. Shvaiko and J. Euzenat. A Survey of Schema Based Matching Approaches. *J. of Data Semantics*, IV - LNCS 3730:146–171, 2005.

[31] W. Su, J. Wang, and F. Lochovsky. Holistic Schema Matching for Web Query Interfaces. In *Proc. of EDBT*, pages 77–94, 2006.

[32] L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data Driven Understanding and Refinement of Schema Mappings. In *Proc. of ACM SIGMOD*, pages 485–496, 2001.