

Finding Semantic Mappings from Relational Tables to Ontologies/Conceptual Models ^{*}

Yuan An¹, Alex Borgida², and John Mylopoulos¹

¹ University of Toronto, Canada

{yuana, jm}@cs.toronto.edu

² Rutgers University, USA

borgida@cs.rutgers.edu

Abstract. Many problems in Information and Data Management require a semantic account of a database schema. At its best, such an account consists of formulas expressing the relationship (“mapping”) between the schema and a formal conceptual model or ontology (CM) of the domain. In this paper we describe the underlying principles, algorithms, and a prototype of a tool which finds such semantic mappings from relational tables to ontologies, when given as input *simple correspondences* from columns of the tables to datatype properties of classes in an ontology. Although the algorithm presented is necessarily heuristic, we offer formal results showing that the answers returned by the tool are “correct” for relational schemas designed according to standard Entity-Relationship techniques. To evaluate its usefulness and effectiveness, we have applied the tool to a number of public domain schemas and ontologies. Our experience shows that significant effort is saved when using it to build semantic mappings from relational tables to ontologies.

Keywords: Semantic interoperability, ontology, mappings.

1 Introduction and Motivation

A number of important database problems have been shown to have improved solutions by using a conceptual model or an ontology (CM) to provide the *precise semantics* of the database schema. These include federated databases, data warehousing [1], and information integration through mediated schemas [10, 5]. (For a survey, see [20].) Since much information on the web is generated from databases (the “deep web”), the recent call for a Semantic Web, which requires a connection between web content and ontologies, provides additional motivation for the problem of associating semantics with data (e.g., [7]). In almost all of these cases semantics of the data is captured by some kind of *semantic mapping* between the database schema and the CM. Although sometimes the mapping is just a *simple* association from terms to terms, in other cases what is required is a *complex* formula, often expressed in logic or a query language [11].

For example, in both the Information Manifold data integration system presented in [10] and the DWQ data warehousing system [1], rules of the form $T(\bar{X})$

^{*} This is an expanded version of an article that was presented at ODBASE’05

$\text{:- } \Phi(\overline{X}, \overline{Y})$ are used to connect a relational data source to a CM described by some Description Logic, where $T(\overline{X})$ is a single predicate representing a table in the relational data source, and $\Phi(\overline{X}, \overline{Y})$ is a conjunctive formula over the predicates representing the concepts and relationships in the CM. In the literature, such a formalism is called local-as-view (LAV), in contrast to global-as-view (GAV), where atomic ontology concepts and properties are specified by queries over the database [11].

In all previous work it has been assumed that *humans* specify the mapping rules – a difficult, time-consuming and error-prone task, especially since the specifier must be familiar with both the semantics of the database schema and the contents of the ontology. As the size and complexity of ontologies increase, it becomes desirable to have some kind of computer tool to assist people in the task. Note that the problem of semantic mapping discovery is superficially similar to that of database schema mapping, but while the goal of the later is finding queries/rules for integrating/translating/exchanging the underlying data, mapping schemas to ontologies is aimed at understanding the semantics of the schemas organized as certain types of structures in terms of a given semantic model. This requires paying special attentions to various semantic constructs in both schemas and ontologies.

In this paper, we propose a tool that assists users in specifying mapping rules between relational database schemas and ontologies by analyzing a standard design process relating the constructs of the relational model with that of conceptual modeling languages. In order to improve the chances of our tool providing useful results, we want to give it some input in addition to the database schema and the ontology. Inspired by the Clio project [14], we therefore ask the tool user to provide *simple correspondences* between atomic elements used in the database schema and those in the ontology such as table columns and datatype properties. Intuitively, the claim is that it is much easier for users to draw simple correspondences/arrows from the column names of the tables in the database to datatype properties of classes in the ontology³ than to compose the logic rules representing the mapping. Given the set of correspondences, the tool is expected to reason about the database schema and the ontology, and to generate a list of candidate rules for each table in the relational database. Ideally, one of the rules is the correct one — capturing the user’s intention underlying the specified correspondences. The following example illustrates the input/output behavior of the tool we seek.

Example 1.1 An ontology contains concepts (classes), attributes of concepts (datatype properties of classes), relationships between concepts (object properties of classes), and cardinality constraints on occurrences of the participating concepts in a relationship. Graphically, we use the UML notations to represent the above information. Figure 1 is an enterprise ontology containing some basic concepts and relationships. Suppose we wish to find the semantics of a relational

³ In fact, there exist already tools used in schema matching which help perform such tasks using linguistic information (e.g., [3, 18]).

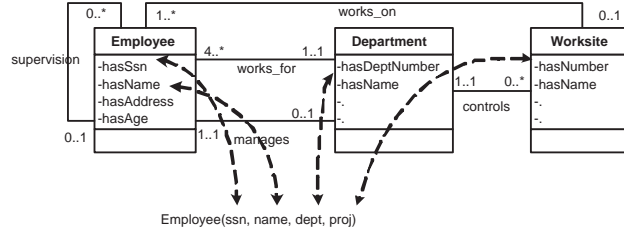


Fig. 1. Relational table, Ontology, and Correspondences.

table $Employee(ssn, name, dept, proj)$ with key ssn in terms of the enterprise ontology. Suppose that by looking at the columns' names and the ontology graph, the user draws the simple correspondences shown as dashed lines in Figure 1. This indicates, for example, that the ssn column corresponds to the $hasSsn$ property of the $Employee$ concept. Using prefixes \mathcal{T} and \mathcal{O} to distinguish tables in the relational schema and concepts in the ontology (both of which will eventually be thought of as predicates), we represent the correspondences as follows:

$\mathcal{T} : Employee.ssn \rightsquigarrow \mathcal{O} : Employee.hasSsn$

$\mathcal{T} : Employee.name \rightsquigarrow \mathcal{O} : Employee.hasName$

$\mathcal{T} : Employee.dept \rightsquigarrow \mathcal{O} : Department.hasDeptNumber$

$\mathcal{T} : Employee.proj \rightsquigarrow \mathcal{O} : Worksite.hasNumber$

Given the above inputs, the tool is expected to produce a list of plausible mapping rules, which would hopefully include the following rule, expressing what we believe to be the semantics of the table:

$\mathcal{T} : Employee(ssn, name, dept, proj) :-$

$\mathcal{O} : Employee(x_1), \mathcal{O} : hasSsn(x_1, ssn), \mathcal{O} : hasName(x_1, name), \mathcal{O} : Department(x_2),$

$\mathcal{O} : works_for(x_1, x_2), \mathcal{O} : hasDeptNumber(x_2, dept), \mathcal{O} : Worksite(x_3), \mathcal{O} : works_on(x_1, x_3),$

$\mathcal{O} : hasNumber(x_3, proj).$

Note that, as explained in [11], the above, admittedly confusing notation in the literature, should really be interpreted as the First Order Logic formula

$(\forall ssn, name, dept, proj) \mathcal{T} : Employee(ssn, name, dept, proj) \Rightarrow$

$(\exists x_1, x_2, x_3) \mathcal{O} : Employee(x_1) \wedge \dots$

because the ontology *explains* what is in the table, rather than guaranteeing that the table will contain all the information from the world is in it. ■

An intuitive (but somewhat naive) solution, inspired by early work of Quillian [17], is based on finding the *shortest* connections between concepts. Technically, this involves (i) finding the minimum spanning tree(s) (actually Steiner trees⁴) connecting the “corresponded concepts” — those that have datatype properties corresponding to table columns, and then (ii) encoding the tree(s) into rules. However, in some cases the spanning/Steiner tree may not provide the desired semantics for a table because of known relational schema design rules. For ex-

⁴ A Steiner tree for set M of nodes in graph G is a minimum spanning tree of M that may contain nodes of G which are not in M .

ample, consider the relational table $Project(name, supervisor)$, with key $name$ corresponding to $\mathcal{O}:Worksite.hasName$, and column $supervisor$ corresponding to $\mathcal{O}:Employee.hasSsn$ in Figure 1. The minimum spanning tree consisting of $Worksite$, $Employee$, and the edge `works_on` probably does not match the semantics of table $Project$ because there are multiple $Employees$ working on a $Worksite$ according to the ontology cardinality, yet the table allows only one to be recorded, since there is a functional dependence from $name$, the key, to $supervisor$. Therefore we must seek a functional connection from $Worksite$ to $Employee$. In this paper, we use ideas of standard relational schema design from ER diagrams in order to systematically uncover the connections between the constructs of relational schemas and those of ontologies. We propose a tool to generate the “reasonable” trees connecting the set of corresponded concepts in an ontology. In contrast to the graph theoretic results which show that there may be too many minimum spanning/Steiner trees among the ontology nodes (for example, there are already 5 minimum spanning trees connecting $Employee$, $Department$, and $Worksite$ in the very simple graph in Figure 1), we expect the tool to generate only a small number of “reasonable” trees.

As mentioned earlier, our approach is directly inspired by the Clio project [14, 15], which developed a successful tool that infers mappings from one set of relational tables and/or XML documents to another, given just a set of correspondences between their respective attributes. Without going into further details at this point, we summarize the contributions which we feel are being made here:

- The paper identifies a new version of the data mapping problem: that of *inferring* complex formulas expressing the semantic mapping between relational database schemas and ontologies from simple correspondences.
- We propose an algorithm to find “reasonable” tree connection(s) in the ontology graph. The algorithm is enhanced to take into account information about the schema (key and foreign key structure), the ontology (cardinality restrictions), and standard database schema design guidelines.
- To gain theoretical confidence, we give formal results which show that if the schema was designed from a CM using techniques well-known in the Entity Relationship literature (which provide a natural semantic mapping and correspondence for each table), then the tool will recover essentially all and only the appropriate semantics. This shows that our heuristics are not just shots in the dark: in the case when the ontology has no extraneous material, and when a table’s schema has not been denormalized, the algorithm will produce good results.
- To test the effectiveness and usefulness of the algorithm in practice, we implemented the algorithm in a prototype tool and applied it to a variety of database schemas and ontologies. Our experience has shown that the user effort in specifying complex mappings by using the tool is significantly less than that by manually writing formulas from scratch.

The rest of the paper is structured as follows. We contrast our approach with related work in section 2, and in Section 3 we present the necessary background and notation. Section 4 describes an intuitive progression of ideas underlying our

approach, while Section 5 provides the mapping inference algorithm. In Section 6 we report on the prototype implementation of these ideas and experience with the prototype. Section 7 shows how to filter out unsatisfied mapping formulas by ontology reasoning, and discusses the issues of generating GAV mapping formulas. Finally, Section 8 concludes and discusses future work.

2 Related Work

The Clio tool [14, 15] discovers formal queries describing how target schemas can be populated with data from source schemas. To compare with it, we could view the present work as extending Clio to the case when the source schema is a relational database while the target is an ontology. For example, in Example 1.1, if one viewed the ontology as a relational schema made of unary tables (such as $Employee(x_1)$), binary tables (such as $hasSsn(x'_1, ssn)$) and the obvious foreign key constraints from binary to unary tables, then one could in fact try to apply directly the Clio algorithm to the problem. The desired mapping formula from Example 1.1 would not be produced for several reasons: (i) Clio [15] does not make a so-called “logical relation” connecting $hasSsn(x'_1, ssn)$ and $hasDeptNumber(x'_2, dept)$, since the chase algorithm of Clio only follows foreign key references *out* of tables. In this particular case, there could only be three logical relations: $Employee \bowtie_{x_1=x'_1} hasSsn$, $Department \bowtie_{x_2=x'_2} hasDeptNumber$, and $works_for \bowtie_{x'_1=x_1} Employee \bowtie_{x'_2=x_2} Department$. (ii) The fact that ssn is a key in the table $T:Employee$, leads us to prefer (see Section 4) a many-to-one relationship, such as $works_for$, over some many-to-many relationship which could have been part of the ontology (e.g., $O:previouslyWorkedFor$); Clio does not differentiate the two. So the work to be presented here analyzes the key structure of the tables and the semantics of relationships (cardinality, IsA) to eliminate/downgrade *unreasonable* options that arise in mapping to ontologies.

Other potentially relevant work includes *data reverse engineering*, which aims to extract a CM, such as an ER diagram, from a database schema. Sophisticated algorithms and approaches to this have appeared in the literature over the years (e.g., [12, 6]). The major difference between data reverse engineering and our work is that we are given an existing ontology, and want to interpret a legacy relational schema in terms of it, whereas data reverse engineering aims to construct a new ontology.

Schema matching (e.g., [3, 18]) identifies semantic relations between schema elements based on their names, data types, constraints, and schema structures. The primary goal is to find the one-to-one simple correspondences which are part of the input for our mapping inference algorithms.

3 Formal Preliminaries

We do not restrict ourselves to any particular language for describing ontologies in this paper. Instead, we use a generic conceptual modeling language (CML), which contains *common* aspects of most semantic data models, UML, ontology

languages such as OWL, and description logics. In the sequel, we use CM to denote an ontology prescribed by the generic CML. Specifically, the language allows the representation of *classes/concepts* (unary predicates over individuals), *object properties/relationships* (binary predicates relating individuals), and *datatype properties/attributes* (binary predicates relating individuals with values such as integers and strings); attributes are single valued in this paper. Concepts are organized in the familiar **is-a** hierarchy. Object properties, and their inverses (which are always present), are subject to constraints such as specification of domain and range, plus cardinality constraints, which here allow 1 as lower bounds (called *total* relationships), and 1 as upper bounds (called *functional* relationships).

We shall represent a given CM using a labeled directed graph, called an *ontology graph*. We construct the ontology graph from a CM as follows: We create a concept node labeled with C for each concept C , and an edge labeled with p from the concept node C_1 to the concept node C_2 for each object property p with domain C_1 and range C_2 ; for each such p , there is also an edge in the opposite direction for its inverse, referred to as p^- . For each attribute f of concept C , we create a separate attribute node denoted as $N_{f,C}$, whose label is f , and add an edge labeled f from node C to $N_{f,C}$.⁵ For each **is-a** edge from a subconcept C_1 to a superconcept C_2 , we create an edge labeled with *is-a* from concept node C_1 to concept node C_2 . For the sake of succinctness, we sometimes use UML notations, as in Figure 1, to represent the ontology graph. Note that in such a diagram, instead of drawing separate attribute nodes, we place the attributes inside the rectangle nodes; and relationships and their inverses are represented by a single undirected edge. The presence of such an undirected edge, labeled p , between concepts C and D will be written in text as $\boxed{C} \text{ ---}p\text{---} \boxed{D}$. If the relationship p is functional from C to D , we write $\boxed{C} \text{ ---}p\text{-->--} \boxed{D}$. For expressive CMLs such as OWL, we may also connect C to D by p if we find an existential restriction stating that each instance of C is related to *some* or *all* instance of D by p .

For relational databases, we assume the reader is familiar with standard notions as presented in [19], for example. We will use the notation $T(\underline{K}, Y)$ to represent a relational table T with columns KY , and key K . If necessary, we will refer to the individual columns in Y using $Y[1], Y[2], \dots$, and use XY as concatenation of columns. Our notational convention is that single column names are either indexed or appear in lower-case. Given a table such as T above, we use the notation $\text{key}(T)$, $\text{nonkey}(T)$ and $\text{columns}(T)$ to refer to K , Y and KY respectively. (Note that we use the terms “table” and “column” when talking about relational schemas, reserving “relation(ship)” and “attribute” for aspects of the CM.) A foreign key (fk) in T is a set of columns F that *references* table T' , and imposes a constraint that the projection of T on F is a subset of the projection of T' on $\text{key}(T')$.

In this paper, a *correspondence* $T.c \leftrightarrow D.f$ will relate column c of table T to attribute f of concept D . Since our algorithms deal with ontology graphs,

⁵ Unless ambiguity arises, we will use “node C ”, when we mean “concept node labeled C ”.

formally a correspondence L will be a mathematical relation $L(T, c, D, f, N_{f,D})$, where the first two arguments determine unique values for the last three.

Finally, for LAV-like mapping, we use Horn-clauses in the form $T(X) :- \Phi(X, Y)$, as described in Section 1, to represent *semantic mappings*, where T is a table with columns X (which become arguments to its predicate), and Φ is a conjunctive formula over predicates representing the CM, with Y existentially quantified, as usual.

4 Principles of Mapping Inference

Given a table T , and correspondences to an ontology provided by a person or a tool, let the set \mathcal{C}_T consist of those concept nodes which have at least one attribute corresponding to some column of T . Our task is to find semantic connections between concepts in \mathcal{C}_T , because attributes can then be connected to the result in the obvious way. The primary principle of our mapping inference algorithm is to look for *smallest* “reasonable” trees connecting nodes in \mathcal{C}_T . We will call such a tree a *semantic tree*.

As mentioned before, the naive solution of finding minimum spanning trees or Steiner trees does not give good enough results, because it must also be “reasonable”. We aim to describe more precisely this notion of “reasonableness”.

Consider the case when $T(\underline{c}, b)$ is a table with key c , corresponding to an attribute f on concept C , and b is a foreign key corresponding to an attribute e on concept B . Then for each value of c (and hence instance of C), T associates at most one value of b (instance of B). Hence the semantic mapping for T should be some formula that acts as a function from its first to its second argument. The semantic trees for such formulas look like functional edges in the ontology, and hence are more reasonable. For example, given table $Dep(\underline{dept}, ssn, \dots)$, and correspondences $\{dept \rightsquigarrow Department.hasDeptNumber, ssn \rightsquigarrow Employee.hasSsn\}$ to the ontology in Figure 1, the proper semantic tree uses $manages^-$ (i.e., $hasManager$) rather than $works_for^-$ (i.e., $hasWorkers$).

Conversely, for table $T'(\underline{c}, b)$, where c and b are as above, an edge that is functional from C to B , or from B to C , is likely not to reflect a proper semantics since it would mean that the key chosen for T' is actually a super-key – an unlikely error. (In our example, consider a table $T(\underline{ssn}, \underline{dept})$, where both columns are foreign keys.)

To deal with such problems, our algorithm will work in two stages: first connecting the concepts corresponding to key columns into a *skeleton tree*, then connecting the rest of the corresponded nodes to the skeleton by functional edges (whenever possible).

We must however also deal with the assumption that the relational schema and the CM were developed independently, which implies that not all parts of the CM are reflected in the database schema. This complicates things, since in building the semantic tree we may need to go through additional nodes, which end up not corresponding to columns of the relational table. For example, consider again

the table $Project(name, supervisor)$ and its correspondences mentioned in Section 1. Because of the key structure of this table, based on the above arguments we will prefer the functional $path\ controls\ \cdot\ manages\ \cdot$ (i.e., `controlledBy` followed by `hasManager`), passing through node *Department*, over the shorter path consisting of edge `works_on`, which is not functional. Similar situations arise when the CM contains detailed *aggregation* hierarchies (e.g., *city* part-of *township* part-of *county* part-of *state*), which are abstracted in the database (e.g., a table with columns for *city* and *state* only).

We have chosen to flesh out the above principles in a systematic manner by considering the behavior of our proposed algorithm on relational schemas designed from Entity Relationship diagrams — a technique widely covered in undergraduate database courses [19]. (We call this *er2rel schema design*.) One benefit of this approach will be to allow us to prove that our algorithm, though heuristic in general, is in some sense “correct” for a certain class of schemas. Of course, in practice such schemas may be “denormalized” in order to improve efficiency, and, as we mentioned, only parts of the CM are realized in the database. We emphasize that our algorithm uses the general principles enunciated above even in such cases, with relatively good results in practice.

To reduce the complexity of the algorithms, which essentially enumerate all trees, and to reduce the size of the answer set, we modify an ontology graph by collapsing multiple edges between nodes E and F , labeled p_1, p_2, \dots say, into a single edge labeled $\langle p_1; p_2; \dots \rangle$. The idea is that it will be up to the user to choose between the alternative labels after the final results have been presented by the tool, though the system may offer suggestions, based on additional information such as heuristics concerning the identifiers labeling tables and columns, and their relationship to property names.

5 Semantic Mapping Inference Algorithms

As mentioned, the algorithm is based in part on the relational database schema design methodology from ER models. We will introduce the details of the algorithm in a gradual manner, by repeatedly adding features of an ER model that appear as part of the CM. We assume that the reader is familiar with basics of ER modeling and database design [19], though we summarize the ideas.

5.1 ER₀: An Initial Subset of ER notions

We start with a subset, ER₀, of ER that supports entity sets E (called just “entity” here), with attributes (referred to by $\text{attribs}(E)$), and binary relationship sets. In order to facilitate the statement of correspondences and theorems, we assume in this section that attributes in the CM have globally unique names. (Our implemented tool does not make this assumption.) An entity is represented as a concept/class in our CM. A binary relationship set corresponds to two properties in our CM, one for each direction. Such a relationship will be called *many-many* if neither it nor its inverse is functional. A *strong entity* S has some attributes

that act as identifier. We shall refer to these using $\text{unique}(S)$ when describing the rules of schema design. A *weak entity* W has instead $\text{localUnique}(W)$ attributes, plus a functional total binary relationship p (denoted as $\text{idRel}(W)$) to an identifying owner entity (denoted as $\text{idOwn}(W)$).

Example 5.1 An ER_0 diagram is shown in Figure 2, which has a weak entity *Dependent* and three strong entities: *Employee*, *Department*, and *Project*. The owner entity of *Dependent* is *Employee* and the identifying relationship is *dependents_of*. Using the notation we introduced, this means that $\text{localUnique}(\text{Dependent}) = \text{deName}$, $\text{idRel}(\text{Dependent}) = \text{dependents_of}$, $\text{idOwn}(\text{Dependent}) = \text{Employee}$. For the owner entity *Employee*, $\text{unique}(\text{Employee}) = \text{hasSsn}$. ■

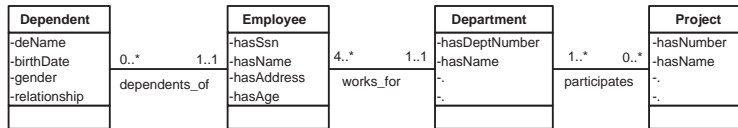


Fig. 2. An ER_0 Example.

Note that information about multi-attribute keys cannot be represented formally in even highly expressive ontology languages such as OWL. So functions like unique are only used while describing the er2rel mapping, and are not assumed to be available during semantic inference. The er2rel design methodology (we follow mostly [12, 19]) is defined by two components. To begin with, Table 1 specifies a mapping $\tau(O)$ returning a relational table schema for every CM component O , where O is either a concept/entity or a binary relationship. (For each relationship exactly one of the directions will be stored in a table.)

In addition to the schema (columns, key, fk's), Table 1 also associates with a relational table $T(V)$ a number of additional notions:

- an *anchor*, which is the central object in the CM from which T is derived, and which is useful in explaining our algorithm (it will be the root of the semantic tree);
- a formula for the semantic mapping for the table, expressed as a rule with head $T(V)$ (this is what our algorithm should be recovering); in the body of the rule, the function $\text{hasAttribs}(x, Y)$ returns conjuncts $\text{attr}_j(x, Y[j])$ for the individual columns $Y[1], Y[2], \dots$ in Y , where attr_j is the attribute name corresponded by column $Y[j]$.
- the formula for a predicate $\text{identify}_C(x, Y)$, showing how object x in (strong or weak) entity C can be identified by values in Y ⁶.

⁶ This is needed in addition to hasAttribs , because weak entities have identifying values spread over several concepts.

ER Model object O	Relational Table $\tau(O)$
Strong Entity S Let $X = \text{attrs}(S)$ Let $K = \text{unique}(S)$	<i>columns:</i> X <i>primary key:</i> K <i>fk's:</i> none <i>anchor:</i> S <i>semantics:</i> $T(X) :- S(y), \text{hasAttrs}(y, X).$ <i>identifier:</i> $\text{identify}_S(y, K) :- S(y), \text{hasAttrs}(y, K).$
Weak Entity W let $E = \text{idOwn}(W)$ $P = \text{idrel}(W)$ $Z = \text{attrs}(W)$ $X = \text{key}(\tau(E))$ $U = \text{localUnique}(W)$ $V = Z - U$	<i>columns:</i> $Z X$ <i>primary key:</i> $U X$ <i>fk's:</i> X <i>anchor:</i> W <i>semantics:</i> $T(X, U, V) :- W(y), \text{hasAttrs}(y, Z), E(w), P(y, w),$ $\text{identify}_E(w, X).$ <i>identifier:</i> $\text{identify}_W(y, U X) :- W(y), E(w), P(y, w), \text{hasAttrs}(y, U),$ $\text{identify}_E(w, X).$
Functional Relationship F $E_1 \text{ --F--> } E_2$ let $X_i = \text{key}(\tau(E_i))$ for $i = 1, 2$	<i>columns:</i> $X_1 X_2$ <i>primary key:</i> X_1 <i>fk's:</i> X_i references $\tau(E_i),$ <i>anchor:</i> E_1 <i>semantics:</i> $T(X_1, X_2) :- E_1(y_1), \text{identify}_{E_1}(y_1, X_1), F(y_1, y_2), E_2(y_2),$ $\text{identify}_{E_2}(y_2, X_2).$
Many-many Relationship M $E_1 \text{ --M-- } E_2$ let $X_i = \text{key}(\tau(E_i))$ for $i = 1, 2$	<i>columns:</i> $X_1 X_2$ <i>primary key:</i> $X_1 X_2$ <i>fk's:</i> X_i references $\tau(E_i),$ <i>semantics:</i> $T(X_1, X_2) :- E_1(y_1), \text{identify}_{E_1}(y_1, X_1), M(y_1, y_2), E_2(y_2),$ $\text{identify}_{E_2}(y_2, X_2).$

Table 1. er2rel Design Mapping.

Note that τ is defined recursively, and will only terminate if there are no “cycles” in the CM (see [12] for definition of cycles in ER).

Example 5.2 When τ is applied to concept *Employee* in Figure 2, we get the table $T:\text{Employee}(\text{hasSsn}, \text{hasName}, \text{hasAddress}, \text{hasAge})$, with the anchor *Employee*, and the semantics expressed by the mapping:

$T:\text{Employee}(\text{hasSsn}, \text{hasName}, \text{hasAddress}, \text{hasAge}) :-$

$\mathcal{O}:\text{Employee}(y), \mathcal{O}:\text{hasSsn}(y, \text{hasSsn}), \mathcal{O}:\text{hasName}(y, \text{hasName}),$

$\mathcal{O}:\text{hasAddress}(y, \text{hasAddress}), \mathcal{O}:\text{hasAge}(y, \text{hasAge}).$

Its identifier is represented by

$\text{identify}_{\text{Employee}}(y, \text{hasSsn}) :- \mathcal{O}:\text{Employee}(y), \mathcal{O}:\text{hasSsn}(y, \text{hasSsn}).$

In turn, $\tau(\text{Dependent})$ produces the table $T:\text{Dependent}(\text{deName}, \text{hasSsn}, \text{birthDate}, \dots)$, whose anchor is *Dependent*. Note that the *hasSsn* column is a foreign key referencing the *hasSsn* column in the $T:\text{Employee}$ table. Accordingly, its semantics is represented as:

$T:\text{Dependent}(\text{deName}, \text{hasSsn}, \text{birthDate}, \dots) :-$

$\mathcal{O}:\text{Dependent}(y)$, $\mathcal{O}:\text{Employee}(w)$, $\mathcal{O}:\text{depends_of}(y,w)$,
 $\text{identify}_{\text{Employee}}(w,\text{hasSsn})$, $\mathcal{O}:\text{deName}(y,\text{deName})$,
 $\mathcal{O}:\text{birthDate}(y,\text{birthDate})\dots$

and its identifier is represented as:

$\text{identify}_{\text{Dependent}}(y,\text{deName},\text{hasSsn}):-$
 $\mathcal{O}:\text{Dependent}(y)$, $\mathcal{O}:\text{Employee}(w)$, $\mathcal{O}:\text{depends_of}(y,w)$,
 $\text{identify}_{\text{Employee}}(w,\text{hasSsn})$, $\mathcal{O}:\text{deName}(y,\text{deName})$.

τ can be applied similarly to the other objects in Figure 5.2. Please note that the anchor of $\tau(\text{works_for})$ is *Employee*, while no single anchor is assigned to $\tau(\text{participates})$. ■

The second step of the er2rel schema design methodology suggests that the schema generated using τ can be modified by (repeatedly) *merging* into the table T_0 of an entity E the table T_1 of some functional relationship involving the same entity E (which has a foreign key reference to T_0). If the semantics of T_0 is $T_0(K, V) :- \phi(K, V)$, and of T_1 is $T_1(K, W) :- \psi(K, W)$, then the semantics of table $T = \text{merge}(T_0, T_1)$ is, to a first approximation, $T(K, V, W) :- \phi(K, V), \psi(K, W)$. And the anchor of T is the entity E . (We defer the description of the treatment of null values which can arise in the non-key columns of T_1 appearing in T .) For example, we could merge the table $\tau(\text{Employee})$ with the table $\tau(\text{works_for})$ in Example 5.2 to form a new table $\mathcal{T}:\text{Employee2}(\text{hasSsn}, \text{hasName}, \text{hasAddress}, \text{hasAge}, \text{hasDeptNumber})$, where the column *hasDeptNumber* is an fk referencing $\tau(\text{Department})$. The semantics of the table is:

$\mathcal{T}:\text{Employee2}(\text{hasSsn}, \text{hasName}, \text{hasAddress}, \text{hasAge}, \text{hasDeptNumber}):-$
 $\mathcal{O}:\text{Employee}(y)$, $\mathcal{O}:\text{hasSsn}(y,\text{hasSsn})$, $\mathcal{O}:\text{hasName}(y,\text{hasName})$,
 $\mathcal{O}:\text{hasAddress}(y, \text{hasAddress})$, $\mathcal{O}:\text{hasAge}(y,\text{hasAge})$,
 $\mathcal{O}:\text{Department}(w)$, $\mathcal{O}:\text{works_for}(y,w)$, $\mathcal{O}:\text{hasDeptNumber}(w,\text{hasDeptNumber})$.

Please note that one conceptual model may result in several different relational schemas, since there are choices in which direction a one-to-one relationship is encoded (which entity acts as a key), and how tables are merged. Note also that the resulting schema is in Boyce-Codd Normal Form, if we assume that the only functional dependencies are those that can be deduced from the ER schema (as expressed in FOL).

In this subsection, we will assume that the CM has no so-called “recursive” relationships relating an entity to itself, and no attribute of an entity corresponds to multiple columns of any table generated from the CM. (We deal with these in Section 5.3.) Note that by the latter assumption, we rule out for now the case when there are several relationships between a weak entity and its owner entity, such as *hasMet* connecting *Dependent* and *Employee*, because in this case $\tau(\text{hasMet})$ will need columns *deName*, *ssn1*, *ssn2*, with *ssn1* helping to identify the dependent, and *ssn2* identifying the (other) employee they met.

Now we turn to the algorithm for finding the semantics of a table in terms of a given CM. It amounts to finding the semantic trees between nodes in the set \mathcal{C}_T singled out by the correspondences from columns of the table T to attributes in the CM. As mentioned previously, the algorithm works in several steps:

1. Determine a skeleton tree connecting the concepts corresponding to key columns; also determine, if possible, a unique anchor for this tree.
2. Link the concepts corresponding to non-key columns using shortest functional paths to the skeleton/anchor tree.
3. Link any unaccounted-for concepts corresponding to other columns by arbitrary shortest paths to the tree.

To flesh out the above steps, we begin with the tables created by the standard design process. If a table is derived by the er2rel methodology from an ER₀ diagram, then Table 1 provides substantial knowledge about how to determine the skeleton tree. However, care must be taken when weak entities are involved. The following example describes the right process to discover the skeleton and the anchor of a weak entity table.

Example 5.3 Consider table $T:Dept(\underline{number}, \underline{univ}, \underline{reportsTo})$, with foreign key (fk) $univ$ referencing table $T:Univ(\underline{name}, \underline{address})$ and correspondences shown in Figure 3. We can tell that $T:Dept$ represents a weak entity since its key has one fk as a subset (referring to the strong entity on which *Department* depends). To find the skeleton and anchor of the table $T:Dept$, we first need to find the skeleton and anchor of the table referenced by the fk $univ$. The answer is $\{University\}$. Next, we should look for a total functional edge (path) from the correspondent of $number$, which is concept *Department*, to the anchor, *University*. As a result, the link $\boxed{Department} \text{ ---belongsTo---} \boxed{University}$ is returned as the skeleton, and *Department* is returned as the anchor. Finally, we can correctly identify the *dean* relationship as the remainder of the connection, rather than the *president* relationship, which would have seemed a superficially plausible alternative to begin with.

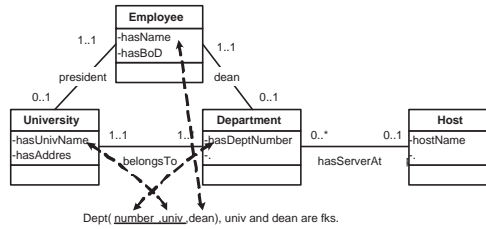


Fig. 3. Finding Correct Skeleton Trees and Anchors.

Furthermore, suppose we need to interpret the table $T:Portal(\underline{dept}, \underline{univ}, \underline{address})$ with the following correspondences:

- $T : Portal.dept \rightsquigarrow \mathcal{O} : Department.hasDeptNumber$
 $T : Portal.univ \rightsquigarrow \mathcal{O} : University.hasUnivName$
 $T : Portal.address \rightsquigarrow \mathcal{O} : Host.hostName,$

where not only is $\{dept, univ\}$ the key but also a fk referencing the key of table $T:Dept$. To find the anchor and skeleton of table $T:Portal$, the algorithm first recursively works on the referenced table. This is also needed when the owner entity of a weak entity is itself a weak entity. ■

The following is the function `getSkeleton` which returns a set of (skeleton, anchor)-pairs, when given a table T and a set of correspondences L from $\text{key}(T)$. The function is essentially a recursive algorithm attempting to reverse the function τ in Table 1. In order to accommodate non-standard tables, the algorithm has branches for finding minimum spanning/Steiner trees as skeletons.

Function `getSkeleton(T, L)`

input: table T , correspondences L for $\text{key}(T)$

output: a set of (skeleton tree, anchor) pairs

steps:

Suppose $\text{key}(T)$ contains fks F_1, \dots, F_n referencing tables $T_1(K_1), \dots, T_n(K_n)$;

1. If $n \leq 1$ and $\text{onc}(\text{key}(T))^7$ is just a singleton set $\{C\}$, then return $(C, \{C\})$.⁸ */*T is likely about a strong entity: base case.*/*
2. Else, let $L_i = \{T_i.K_i \leftrightarrow L(T, F_i)\}$ */*translate corresp's thru fk reference.*/*;
 compute $(Ss_i, Anc_i) = \text{getSkeleton}(T_i, L_i)$.
 - (a) If $\text{key}(T) = F_1$, then return (Ss_1, Anc_1) . */*T looks like the table for the functional relationship of a weak entity.*/*
 - (b) If $\text{key}(T) = F_1A$, where columns A are not part of a foreign key then */*T is possibly a weak entity*/*
 if $Anc_1 = \{N_1\}$ and $\text{onc}(A) = \{N\}$ such that there is a (shortest) total functional path π from N to N_1 , then return $(\text{combine}^9(\pi, Ss_1), \{N\})$.
*/*N is a weak entity.*/*
 - (c) Else suppose $\text{key}(T)$ has non-fk columns $A[1], \dots, A[m]$, ($m \geq 0$); let $Ns = \{Anc_i\} \cup \{\text{onc}(A[j]), j = 1, \dots, m\}$; find skeleton tree S' connecting the nodes in Ns where any pair of nodes in Ns is connected by a (shortest) many-many path; return $(\text{combine}(S', \{Ss_j\}), Ns)$. */*Deals with many-to-many binary relationships; also the default action for non-standard cases, such as when not finding identifying relationship from a weak entity to the supposed owner entity. In this case no unique anchor exists.*/*

In order for `getSkeleton` to terminate, it is necessary that there be no cycles in fk references in the schema. Such cycles (which may have been added to represent additional integrity constraints, such as the fact that a property is total) can be eliminated from a schema by replacing the tables involved with their outer join over the key. `getSkeleton` deals with strong entities and their functional relationships in step (1), with weak entities in step (2.b), and so far, with functional relationships of weak entities in (2.a). In addition to being a catch-all, step (2.c) deals with tables representing many-many relationships

⁷ $\text{onc}(X)$ is the function which gets the set M of concepts corresponded by the columns X .

⁸ Both here and elsewhere, when a concept C is added to a tree, so are edges and nodes for C 's attributes that appear in L .

⁹ Function `combine` merges edges of trees into a larger tree.

(which in this section have key $K = F_1F_2$), by finding anchors for the ends of the relationship, and then connecting them with paths that are not functional, even when every edge is reversed.

To find the entire semantic tree of a table T , we must connect the concepts corresponded by the rest of the columns, i.e., $\text{nonkey}(T)$, to the anchor. The connections should be (shortest) functional edges (paths), since the key determines at most one value for them; however, if such a path cannot be found, we use an arbitrary shortest path. The following function, `getTree`, achieves the goal.

Function `getTree(T,L)`

input: table T , correspondences L for $\text{columns}(T)$

output: set of semantic trees ¹⁰

steps:

1. Let L_k be the subset of L containing correspondences from $\text{key}(T)$;
compute $(S', Anc') = \text{getSkeleton}(T, L_k)$.
2. If $\text{onc}(\text{nonkey}(T)) - \text{onc}(\text{key}(T))$ is empty, then return (S', Anc') . */*if all columns correspond to the same set of concepts as the key does, then return the skeleton tree.*/*
3. For each foreign key F_i in $\text{nonkey}(T)$ referencing $T_i(K_i)$:
let $L_k^i = \{T_i.K_i \rightsquigarrow L(T, F_i)\}$, and compute $(Ss_i'', Anc_i'') = \text{getSkeleton}(T_i, L_k^i)$. */*recall that the function $L(T, F_i)$ is derived from a correspondence $L(T, F_i, D, f, N_{f,D})$ such that it gives a concept D and its attribute f ($N_{f,D}$ is the attribute node in the ontology graph.)*/*
find $\pi_i = \text{shortest functional path from } Anc' \text{ to } Anc_i''$; let $S = \text{combine}(S', \pi_i, \{Ss_i''\})$.
4. For each column c in $\text{nonkey}(T)$ that is not part of an fk, let $N = \text{onc}(c)$; find $\pi = \text{shortest functional path from } Anc' \text{ to } N$; update $S := \text{combine}(S, \pi)$.
5. In all cases above asking for functional paths, use a shortest path if a functional one does not exist.
6. Return S .

The following example illustrates the use of `getTree` when seeking to interpret a table using a different CM than the one originally.

Example 5.4 In Figure 4, the table $T: \text{Assignment}(\text{emp}, \text{proj}, \text{site})$ was originally derived from a CM with the entity *Assignment* shown on the right-hand side of the vertical dashed line. To interpret it by the CM on the left-hand side, the function `getSkeleton`, in Step 2.c, returns `Employee` ---assignedTo--- `Project` as the skeleton, and the function `getTree` connects *Worksite* to *Employee* via *works_on* in Step 4, to build the final semantic tree. ■

To get the logic formula from a tree based on correspondence L , we provide the procedure `encodeTree(S, L)` below, which basically assigns variables to nodes, and connects them using edge labels as predicates.

¹⁰ To make the description simpler, at times we will not explicitly account for the possibility of multiple answers. Every function is extended to set arguments by element-wise application of the function to set members.

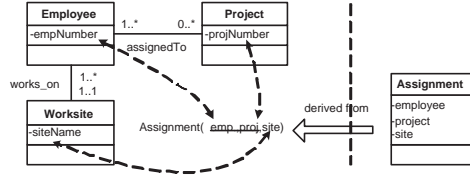


Fig. 4. Independently Developed Table and CM.

Function encodeTree(S, L)

input: subtree S of ontology graph, correspondences L from table columns to attributes of concept nodes in S .

output: variable name generated for root of S , and conjunctive formula for the tree.

steps: Suppose N is the root of S . Let $\Psi = true$.

1. if N is an attribute node with label f , find d such that $L(-, d, -, f, N) = true$, return($d, true$). /*for leaves of the tree, which are attribute nodes, return the corresponding column name as the variable and the formula true.*/

2. if N is a concept node with label C , then introduce new variable x ; add conjunct $C(x)$ to Ψ ;

for each edge p_i from N to N_i /*recursively get the subformulas.*/

let S_i be the subtree rooted at N_i ,

let $(v_i, \phi_i(Z_i)) = encodeTree(S_i, L)$,

add conjuncts $p_i(x, v_i) \wedge \phi_i(Z_i)$ to Ψ ;

3. return (x, Ψ) .

Example 5.5 Figure 5 is the fully specified semantic tree returned by the algorithm for the $T:Dept(number, univ, dean)$ table in Example 5.3. Taking *Department* as the root of the tree, function encodeTree generates the following formula:

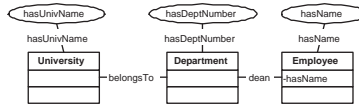


Fig. 5. Semantic Tree for Dept Table.

$Department(x), hasDeptNumber(x, number), belongsTo(x, v_1), University(v_1), hasUnivName(v_1, univ), dean(x, v_2), Employee(v_2), hasName(v_2, dean)$.

As expected, the formula is the semantics the table $T:Dept$ as assigned by the er2rel design τ . ■

Now we turn to the properties of the mapping algorithm. In order to be able to make guarantees, we have to limit ourselves to “standard” relational schemas, since otherwise the algorithm cannot possibly guess the intended meaning of an arbitrary table. For this reason, let us consider only schemas generated by the er2rel methodology from a CM encoding a ER diagram. We are interested in two properties: (1) A sense of “completeness”: the algorithm finds the correct

semantics (as specified in Table 1). (2) A sense of “soundness”: if for such a table there are multiple semantic trees returned by the algorithm, then each of the trees would produce an indistinguishable relational table according to the `er2rel` mapping. (Note that multiple semantic trees are bound to arise when there are several relationships between 2 entities which cannot be distinguished semantically in a way which is apparent in the table (e.g., 2 or more functional properties from A to B). To formally specify the properties, we have the following definitions.

A *homomorphism* h from the columns of a table T_1 to the columns of a table T_2 is a one-to-one mapping $h: \text{columns}(T_1) \rightarrow \text{columns}(T_2)$, such that i) $h(c) \in \text{key}(T_2)$ for every $c \in \text{key}(T_1)$; ii) $h(c)$ is a fk of T_2 for every c which is a fk of T_1 ; and iii) if c is a fk of T_1 , then there is a homomorphism from the $\text{key}(T'_1)$ of T'_1 referenced by c to the $\text{key}(T'_2)$ of T'_2 referenced by $h(c)$ in T_2 .

Definition 1. A relational table T_1 is isomorphic to another relational table T_2 , if there is a homomorphism from $\text{columns}(T_1)$ to $\text{columns}(T_2)$ and vice versa.

Informally, two tables are isomorphic if there is a bijection between their columns which preserves recursively the key and foreign key structures. These structures have direct connections with the structures of the ER diagrams from which the tables were derived. Since the `er2rel` mapping τ may generate the “same” table when applied to different ER diagrams (considering attribute/column names have been handled by correspondences), a mapping discovery algorithm with “good” properties should report all and only those ER diagrams.

To specify the properties of the algorithm, suppose that the correspondence L_{id} is the identity mapping from table columns to attribute names, as set up in Table 1. The following lemma states the interesting property of `getSkeleton`.

Lemma 1. Let ontology graph \mathcal{G} encode an ER_0 diagram \mathcal{E} . Let $T = \tau(C)$ be a relational table derived from an object C in \mathcal{E} according to the `er2rel` rules in Table 1. Given L_{id} from T to \mathcal{G} , and $L' = \text{the restriction of } L_{id} \text{ to } \text{key}(T)$, then `getSkeleton`(T, L') returns (S, Anc) such that,

- Anc is the anchor of T ($\text{anchor}(T)$).
- If C corresponds to a (strong or weak) entity, then $\text{encodeTree}(S, L')$ is logically equivalent to identify_C .

Proof Sketch The lemma is proven by using induction on the number of applications of `getSkeleton` on the table T .

At the base case, step 1 of `getSkeleton` indicates that $\text{key}(T)$ links to a single concept in \mathcal{G} . According to the `er2rel` design, table T is derived either from a strong entity or a functional relationship from a strong entity. For either case, $\text{anchor}(T)$ is the strong entity, and $\text{encodeTree}(S, L')$ is logically equivalent to identify_E , where E is the strong entity.

For the induction hypothesis, we assume that the lemma holds for each table that is referenced by a foreign key in T .

On the induction steps, step 2.(a) identifies that table T is derived from a functional relationship from a weak entity. By the induction hypothesis, the lemma holds for the weak entity. So does it for the relationship.

Step 2.(b) identifies that T is a table representing a weak entity W with an owner entity E . Since there is only one total functional relationship from a weak entity to its owner entity, `getSkeleton` correctly returns the identifying relationship. By the induction hypothesis, we prove that `encodeTree(S, L')` is logically equivalent to `identify $_{\mathcal{W}}$` . ■

We now state the desirable properties of the mapping discovery algorithm. First, `getTree` finds the desired semantic mapping, in the sense that

Theorem 1. *Let ontology graph \mathcal{G} encode an ER_0 diagram \mathcal{E} . Let table T be part of a relational schema obtained by `er2rel` derivation from \mathcal{E} . Given L_{id} from T to \mathcal{G} , then some tree S returned by `getTree(T, L_{id})` has the property that the formula generated by `encodeTree(S, L_{id})` is logically equivalent to the semantics assigned to T by the `er2rel` design.*

Proof Sketch By Lemma 1, it is easy to prove the four cases listed in Table 1, so we skip it.

Now suppose T is the table by merging an entity table with tables representing functional relationships involving the same entity, then we use induction on the number of merged tables to prove the theorem.

The base case is that there is only one functional relationship table that is merged. Step 3 of `getTree` returns the functional relationship as one of resulted semantic trees.

On the induction steps, step 3 of `getTree` will return the semantic tree, S , which consists of the $k + 1$ functional relationships. Pick any one of the $k + 1$ functional relationships, and the rest tree consists of k functional relationship. By induction hypothesis, encoding the rest tree is logically equivalent to the semantics of the table by merging the k functional relationship tables with the entity table. Then, we prove it is also true for $k + 1$. ■

Note that this result is non-trivial, since, as explained earlier, it would not be satisfied by the current Clio algorithm [15], if applied blindly to \mathcal{E} viewed as a relational schema with unary and binary tables. Since `getTree` may return multiple answers, the following converse “soundness” result is significant.

Theorem 2. *If S' is any tree returned by `getTree(T, L_{id})`, with T, L_{id} , and \mathcal{E} as above in Theorem 1, then the formula returned by `encodeTree(S', L_{id})` represents the semantics of some table T' derivable by `er2rel` design from \mathcal{E} , where T' is isomorphic to T .*

Proof Sketch The theorem is proven by proving that each tree returned by `getTree` will result in an isomorphic table T' .

For the four cases in Table 1, `getTree` will return a single semantic tree for an entity (strong or weak) table and (probably) multiple semantic trees for a (functional) relationship table. Each of the semantic trees returned for a relationship table is identical to the original ER diagram in terms of the shape and the cardinality constraints. As a result, applying τ to the semantic tree generates an isomorphic table to T .

Now suppose T is the table by merging an entity table with n tables representing n functional relationships from the entity to other n entities. `getTree` will

return semantic trees each of which consists of n functional relationships from E . By the `er2rel` and `merge` operation, mapping any one of the semantic tree will obtain a table T' which is isomorphic to T ■

We wish to point out that the above algorithm performs in a reasonable manner even on common non-standard relational schemas. Example 5.4 has illustrated this point already. Here, we examine an alternate example: Consider the relational table $T(\overline{personName}, \overline{cityName}, \overline{countryName})$, where the columns correspond to, respectively, attributes $pname$, $cname$, and $ctrname$ of concepts $Person$, $City$ and $Country$ in a CM. If the CM contains a path such that $\boxed{Person} \text{ -- bornIn --> } \boxed{City} \text{ -- locatedIn --> } \boxed{Country}$, then the above table, which is not in 3NF and was not obtained using `er2rel` design (which would have required a table for $City$), would still get the proper semantics:

$T(\overline{personName}, \overline{cityName}, \overline{countryName}) :-$
 $Person(x_1), City(x_2), Country(x_3), \text{bornIn}(x_1, x_2), \text{locatedIn}(x_2, x_3),$
 $pname(x_1, \overline{personName}), cname(x_2, \overline{cityName}), ctrname(x_3, \overline{countryName}).$

If, on the other hand, there was a shorter functional path from $Person$ to $Country$, say an edge labeled `citizenOf`, then the mapping suggested would have been:

$T(\overline{personName}, \overline{cityName}, \overline{countryName}) :-$
 $Person(x_1), City(x_2), Country(x_3), \text{bornIn}(x_1, x_2), \text{citizenOf}(x_1, x_3), \dots$

which corresponds to the `er2rel` design. Moreover, had `citizenOf` not been functional, then once again the semantics produced by the algorithm would correspond to the non-3NF interpretation, which is reasonable since the table, having only $\overline{personName}$ as key, could not store multiple country names for a person.

5.2 ER₁: Reified Relationships

It is desirable to also have n-ary relationship sets connecting entities, and to allow relationship sets to have attributes in an ER model; we label the language allowing us to model such aspects by ER₁. Unfortunately, these features are not directly supported in most CMLs, such as OWL, which only have binary relationships. Such notions must instead be represented by “reified relationships” [2] (we use an annotation `*` to indicate the reified relationships in a diagram): concepts whose instances represent tuples, connected by so-called “roles” to the tuple elements. So, if $Buys$ relates $Person$, $Shop$ and $Product$, through roles $buyer$, $source$ and $object$, then these are explicitly represented as (functional) binary associations, as in Figure 6. And a relationship attribute, such as when the buying occurred, becomes an attribute of the $Buys$ concept, such as $whenBought$.

Unfortunately, reified relationships cannot be distinguished reliably from ordinary entities in normal CMLs based on purely formal, syntactic grounds, yet

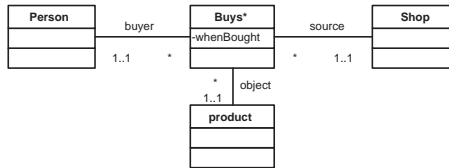


Fig. 6. N-ary Relationship Reified.

“reified relationships” [2] (we use an annotation `*` to indicate the reified relationships in a diagram): concepts whose instances represent tuples, connected by so-called “roles” to the tuple elements. So, if $Buys$ relates $Person$, $Shop$ and $Product$, through roles $buyer$, $source$ and $object$, then these are explicitly represented as (functional) binary associations, as in Figure 6. And a relationship attribute, such as when the buying occurred, becomes an attribute of the $Buys$ concept, such as $whenBought$.

they need to be treated in special ways during semantic recovery. For this reason we assume that they can be distinguished on *ontological grounds*. For example, in Dolce [4], they are subclasses of top-level concepts *Quality* and *Perdurant/Event*. For a reified relationship R , we use functions $\text{roles}(R)$ and $\text{attrs}(R)$ to retrieve the appropriate (binary) properties.

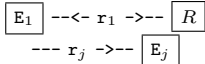
ER model object O	Relational Table $\tau(O)$
Reified Relationship R if there is a functional role r_1 for R  let $Z = \text{attrs}(R)$ $X_i = \text{key}(\tau(E_i))$ where E_i fills role r_i	<i>columns:</i> $ZX_1 \dots X_n$ <i>primary key:</i> X_1 <i>fk's:</i> X_1, \dots, X_n <i>anchor:</i> R <i>semantics:</i> $T(ZX_1 \dots X_n) :- R(y), E_i(w_i), \text{hasAttrs}(y, Z), r_i(y, w_i), \text{identify}_{E_i}(w_i, X_i), \dots$ <i>identifier:</i> $\text{identify}_R(y, X_1) :- R(y), E_1(w), r_1(y, w), \text{identify}_{E_1}(w, X_1).$
Reified Relationship R if r_1, \dots, r_n are roles of R let $Z = \text{attrs}(R)$ $X_i = \text{key}(\tau(E_i))$ where E_i fills role r_i	<i>columns:</i> $ZX_1 \dots X_n$ <i>primary key:</i> $X_1 \dots X_n$ <i>fk's:</i> X_1, \dots, X_n <i>anchor:</i> R <i>semantics:</i> $T(ZX_1 \dots X_n) :- R(y), E_i(w_i), \text{hasAttrs}(y, Z), r_i(y, w_i), \text{identify}_{E_i}(w_i, X_i), \dots$ <i>identifier:</i> $\text{identify}_R(y, \dots X_i \dots) :- R(y), \dots E_i(w_i), r_i(y, w_i), \text{identify}_{E_i}(w_i, X_i), \dots$

Table 2. er2rel Design for Reified Relationship.

The er2rel design τ of relational tables for reified relationships is an extension of the treatment of binary relationships, and is shown in Table 2. As with entity keys, we are unable to capture in CM situations where some subset of more than one roles uniquely identifies the relationship. The er2rel design τ on ER₁ also admits the *merge* operation on tables generated by τ . Merging applies to an entity table with other tables of some functional relationships involving the same entity. In this case, the merged semantics is the same as that of merging tables obtained by applying τ to ER₀, with the exception that some functional relationships may be reified.

To discover the correct anchor for reified relationships and get the proper tree, we need to modify `getSkeleton`, by adding the following case between steps 2(b) and 2(c):

- If $\text{key}(T) = F_1 F_2 \dots F_n$ and there exist reified relationship R with n roles r_1, \dots, r_n pointing at the singleton nodes in Anc_1, \dots, Anc_n respectively, then let $S = \text{combine}(\{r_j\}, \{Ss_j\})$, and return $(S, \{R\})$.

`getTree` should compensate for the fact that if `getSkeleton` finds a *reified* version of a many-many binary relationship, it will no longer look for an unreified one in step 2c. So after step 1. we add

- if $\text{key}(T)$ is the concatenation of two foreign keys F_1F_2 , and $\text{nonkey}(T)$ is empty, compute (S_{s_1}, Anc_1) and (S_{s_2}, Anc_2) as in step 2. of `getSkeleton`; then find ρ =shortest many-many path connecting Anc_1 to Anc_2 ; return $(S') \cup (\text{combine}(\rho, S_{s_1}, S_{s_2}))$

In addition, when traversing the ontology graph for finding shortest paths in both functions, we need to recalculate the lengths of paths when reified relationship nodes are present. Specifically, a path of length 2 passing through a reified relationship node should be counted as a path of length 1. Note that a semantic tree that includes a reified relationship node is valid only if all roles of the reified relationship have been included in the tree. Moreover, if the reified relation had attributes of its own, they would show up as columns in the table that are not part of any foreign key. Therefore, a filter is required at the last stage of the algorithm:

- If a reified relationship R appears in the final semantic tree, then so must all its role edges. And if one such R has as attributes the columns of the table which do not appear in foreign keys or the key, then all other candidate semantics need to be eliminated.

The previous version of `getTree` was set up so that with these modifications, roles and attributes to reified relationships will be found properly.

If we continue to assume that no more than one column corresponds to each entity attribute, the previous theorems hold for ER_1 as well. To see this, consider the following two points. First, the appropriate tree is identified for any table generated from a reified relationship, since the foreign keys of the table identify exactly the participants in the relationship. Second, if an entity E has a set of (binary) functional relationships connecting to a set of entities E_1, \dots, E_n , then merging the corresponding tables with $\tau(E)$ results in a table that is isomorphic to a reified relationship table, where the reified relationship has a single functional role with filler E and all other role fillers are the set of entities E_1, \dots, E_n .

5.3 Replication

We next deal with the full ER_1 model, by allowing recursive relationships, where a single entity plays multiple roles, and the merging of tables for different functional relationships connecting the same pair of concepts (e.g., `works_for` and `manages`). In such cases the mapping described in Table 1 is not quite correct because column names will be repeated in the multiple occurrences of the foreign keys. We will distinguish these (again, for ease of presentation) by adding superscripts as needed. For example, if entity set *Person*, with key *ssn*, is connected to itself by the *likes* property, then the table for *likes* will have schema $T[\underline{ssn}^1, \underline{ssn}^2]$.

During mapping discovery, such situations are signaled by the presence of multiple columns c and d of table T corresponding to the same attribute f of concept C . In such situations, we modify the algorithm to first make a copy C_{copy}

of node C , as well as its attributes, in the ontology graph. C_{copy} participates in all the object relations C did, so edges for this must also be added. After replication, we can set $\text{onc}(c) = C$ and $\text{onc}(d) = C_{copy}$, or $\text{onc}(d) = C$ and $\text{onc}(c) = C_{copy}$ (recall that $\text{onc}(c)$ retrieves the concept corresponded to by column c in the algorithm). This ambiguity is actually required: given a CM with $Person$ and $likes$ as above, a table $T[\underline{ssn}^1, \underline{ssn}^2]$ could have two possible semantics: $likes(ssn^1, ssn^2)$ and $likes(ssn^2, ssn^1)$, the second one representing the inverse relationship, $likedBy$. The problem arises not just with recursive relationships: consider the case of a table $T[\underline{ssn}, \underline{addr}^1, \underline{addr}^2]$, where $Person$ is connected by two relationships, $home$ and $office$, to concept $Building$, which has an $address$ attribute.

The main modification needed to the `getSkeleton` and `getTree` algorithms is that no tree should contain both a functional edge $\boxed{D} \text{ --- } p \text{ ->--- } \boxed{C}$ and its replicate $\boxed{D} \text{ --- } p \text{ ->--- } \boxed{C_{copy}}$, (or several replicates), since a function has a single value, and hence the different columns of a tuple will end up having identical values: a clearly poor schema.

The immediate question after the modification is that do the properties as specified in Theorem 1 and Theorem 2 still hold? The answer is that the property of “completeness” specified as in Theorem 1 still holds, while the property of “soundness” as specified in Theorem 2 holds on the condition that if there is only one total functional relationship from a weak entity to its owner entity. Let us see why this is the case. If an attribute of an entity set is corresponded by more than one column in a table, then it indicates that the table encodes multiple relationships/roles participated by the single entity. Notice that our algorithm is to find a tree structure from an ontology graph. Therefore, multiple paths passing through a single node (not the root) cannot be contained in a tree structure simultaneously. By duplicating the concept node as well as all the relationships it participates, we create an new ontology graph which separates the different relationships into different nodes that actually are clones of the same entity. Furthermore, by not allowing a functional relationship and its duplication to be included in the same tree, we have kept the original semantics of the functional relationship. These lines of reasoning give rise to the first property. For the second property, the following example gives a counterexample.

Example 5.6 An educational department in a provincial government records the transfers of students between universities in its databases. A student is a weak entity depending for identification on the university in which the student is currently registered. A transferred student must have registered in another university before transferring. The table $\mathcal{T}:\underline{Transferred}(\underline{sno}, \underline{univ}, \underline{sname})$ records who are the transferred students, and their name. The table $\mathcal{T}:\underline{previous}(\underline{sno}, \underline{univ}, \underline{pUniv})$ stores the information about the $previousUniv$ relationship. A CM is depicted in Figure 7. To discover the semantics of table $\mathcal{T}:\underline{Transferred}$, we link the columns to the attributes in the CM as shown in Figure 7. One of the skeletons returned by the algorithm for the $\mathcal{T}:\underline{Transferred}$ will be $\boxed{\underline{TransferredStudent}}$ --- $\underline{previousUniv}$ ->--- $\boxed{\underline{University}}$. But the design resulting from this ac-

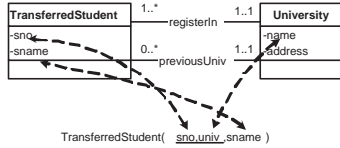


Fig. 7. Multiple Total Functional Relationships from WE to its Owner Entity.

According to the $er2rel$ mapping is not isomorphic to $key(Transferred)$, since $previousUniv$ is not the identifying relationship of the weak entity $TransferredStudent$. ■

From above example, we can see that the problem is the inability of CMLs like UML and OWL to fully capture notions like “weak entity” (specifically the notion of identifying relationship), which play a role in ER-based design. We expect such cases to be quite rare though – we certainly have not encountered any in our example databases.

5.4 Extended ER: Adding Class Specialization

The ability to represent subclass hierarchies, such as the one in Figure 8 is a hallmark of CMLs and modern so-called Extended ER (EER) modeling.

Almost all textbooks (e.g., [19]) describe several techniques for designing relational schemas in the presence of class hierarchies

1. Map each concept/entity into a separate table following the standard $er2rel$ rules. This approach requires two adjustments: First, subclasses must inherit identifying attributes from a single super-class, in order to be able to generate keys for their tables. Second, in the table created for an immediate subclass C' of class C , its key $key(\tau(C'))$ should also be set to reference as a foreign key $\tau(C)$, as a way of maintaining inclusion constraints dictated by the is-a relationship.
2. Expand inheritance, so that *all* attributes and relations involving a class C appear on all its subclasses C' . Then generate tables as usual for the subclasses C' , though not for C itself. This approach is used only when the subclasses cover the superclass.
3. Some researchers also suggest a third possibility: “Collapse up” the information about subclasses into the table for the superclass. This can be viewed as the result of $merge(T_C, T_{C'})$, where $T_C(K, A)$ and $T_{C'}(K, B)$ are the tables generated for C and its subclass C' according to technique (1.) above. In order for this design to be “correct”, [12] requires that $T_{C'}$ not be the target of any foreign key references (hence not have any relationships mapped to tables), and that B be non-null (so that instances of C' can be distinguished from those of C).

The use of the key for the root class, together with inheritance and the use of foreign keys to also check inclusion constraints, make many tables highly am-

biguous. For example, according to the above, table $T(\underline{ss\#}, crsId)$, with $ss\#$ as the key and a foreign key referencing T' , could represent at least

- (a) *Faculty teach Course*
- (b) *Lecturer teach Course*
- (c) *Lecturer coord Course*.

This is made combinatorially worse by the presence of multiple and deep hierarchies (e.g., imagine a parallel *Course* hierarchy), and the fact that not all ontology concepts are realized in the database schema, according to our scenario. For this reason, we have chosen to try to deal with some of the ambiguity by relying on users, during the establishment of correspondences. Specifically, the user is supposed to provide a correspondence from column c to attribute f on the lowest class whose instances provide data appearing in the column. Therefore, in the above example of table $T(\underline{ss\#}, crsId)$, $ss\#$ should be set to correspond to ssn on *Faculty* in case (a), while in cases (b) and (c) it should correspond to $ss\#$ on *Lecturer*. This decision was also prompted by the CM manipulation tool that we are using, which automatically expands inheritance, so that $ss\#$ appears on all subclasses.

Under these circumstances, in order to capture designs (1.) and (2.) above, we do not need to modify our earlier algorithm in any way, if we first expand inheritance in the graph. So the graph would show `Lecturer` -- teaches;coord --> `Course` in the above example, and *Lecturer* would have all the attributes of *Faculty*.

To handle design (3.), we can add to the graph an actual edge for the inverse of the **is-a** relation: a functional edge labeled *alsoA*, with lower-bound 0: `C` --- alsoA --> `C'`, connecting superclass C to each of its subclasses C' . It is then sufficient to allow functional paths between concepts to consist of *alsoA* edges, in addition to the normal kind, in `getTree`. Furthermore, only edges of normal kind of relationship account for the length of a path.

In terms of the properties of the algorithm we have been considering so far, the above three paragraphs have explained that among the answers returned by the algorithm will be the correct one. On the other hand, if there are multiple results returned by the algorithm, as shown in Example 5.6, some semantic trees may not result in isomorphic tables to the original table, if there are more than one total functional relationships from a weak entity to its owner entity.

5.5 Outer Joins

The observant reader has probably noticed that the definition of the semantic mapping for $T = \text{merge}(T_E, T_p)$ was not quite correct: $T(\underline{K}, V, W) :- \phi(K, V), \psi$

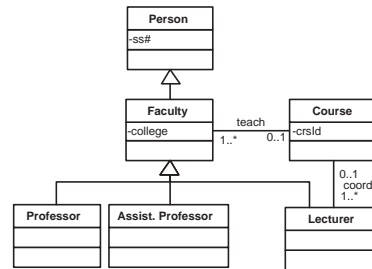


Fig. 8. Specialization Hierarchy.

(K, W) describes a join on K , rather than a left-outer join, which is what is required if p is a non-total relationship. In order to specify the equivalent of outer joins in a perspicuous manner, we will use conjuncts of the form $[\mu(X, Y)]^Y$, which will stand for the formula $\mu(X, Y) \vee (Y = null \wedge \neg \exists Z. \mu(X, Z))$, indicating that null should be used if there are no satisfying values for the variables Y . With this notation, the proper semantics for merge is $T(\underline{K}, V, W) : -\phi(K, V), [\psi(K, W)]^W$.

In order to obtain the correct formulas from trees, `encodeTree` needs to be modified so that when traversing a non-total edge p_i that is not part of the skeleton, in the second-to-last line of the algorithm we must allow for the possibility of v_i not existing.

6 Implementation and Experience

So far, we have developed the mapping inference algorithm by investigating the connections between the semantic constraints in relational models and that in ontologies. The theoretical results show that our algorithm will report the “right” semantics for most schemas designed following the widely accepted design methodology. Nonetheless, it is crucial to test the algorithm in real-world schemas and ontologies to see its overall performance. To do this, we have implemented the mapping inference algorithm in our prototype system MAPONTO, and have applied it on a set of schemas and ontologies. In this section, we describe the implementation and provide some evidence for the effectiveness and usefulness of the prototype tool by discussing the set of experiments and our experience.

Implementation. We have implemented the MAPONTO tool as a third-party plugin of the well-known KBMS *Protégé*¹¹ which is an open platform for ontology modeling and knowledge acquisition. As OWL becomes the official ontology language of the W3C, intended for use with Semantic Web initiatives, we use OWL as the CML in the tool. This is also facilitated by the *Protégé*’s OWL plugin [9], which can be used to edit OWL ontologies, to access reasoners for them, and to acquire instances for semantic markup. The MAPONTO plugin is implemented as a full-size user interface tab that takes advantage of the views of *Protégé* user interface. As shown in Figure 9, users can choose database schemas and ontologies, create and manipulate correspondences, generate and edit candidate mapping formulas and graphical connections, and produce and save the final mappings into designated files. Our decision to build our system on top of *Protégé* was driven by various factors. First, since MAPONTO acts as a bridge between database schemas and ontologies, and ontologies tend to be more complex, it was more straight-forward to take an existing ontology development environment as a starting point. Second, in addition to the collection of standard plugins for editing ontology classes, properties, forms and instances, a library of other plugins exist that visualize ontologies graphically and manage ontology versions.

¹¹ <http://protege.stanford.edu>

Those plugins sustain our goal of providing an interactively intelligent tool to database administrators so that they may establish semantic mappings from the database to ontologies more effectively.

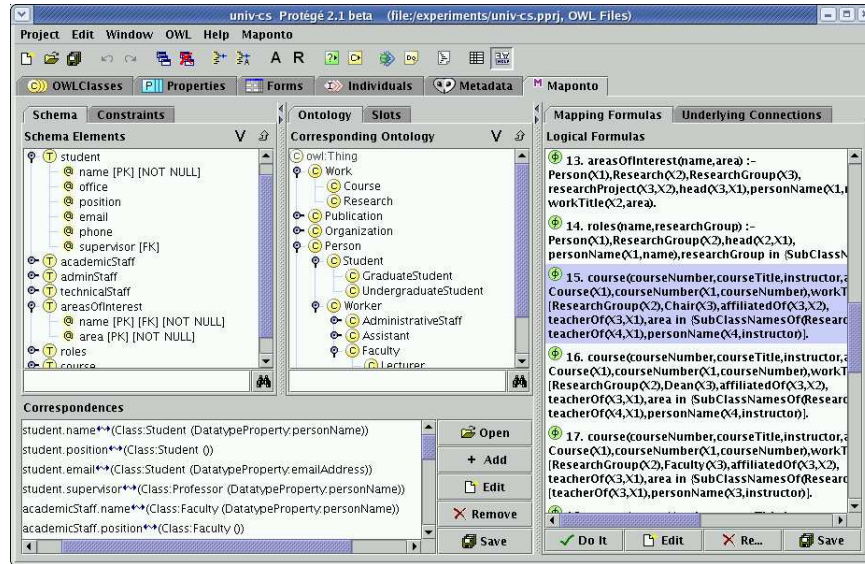


Fig. 9. MAPONTO plugin of Protege.

Schemas and Ontologies. Our test data were obtained from various sources, and we have ensured that the databases and ontologies were developed independently. The test data are listed in Table 3. They include the following databases: the Department of Computer Science database in University of Toronto; the VLDB conference database; the DBLP computer science bibliography database; the COUNTRY database appearing in one of reverse engineering papers [8] (Although the *country* schema is not a real-world database, it appears as a complex experimental example in [8], and has some reified relationship tables, so we chose it to test this aspect of our algorithm); and the test schemas in OBSERVER [13] project. For the ontologies, our test data include: the academic department ontology in the DAML library; the academic conference ontology from the SchemaWeb ontology repository; the bibliography ontology in the library of the Stanford’s Ontolingua server; and the CIA factbook ontology. Ontologies are described in OWL. For each ontology, the number of links indicates the number of edges in the multi-graph resulted from object properties. We have made all these schemas and ontologies available on our web page: www.cs.toronto.edu/~yuana/research/maponto/relational/testData.html.

Results and Experience. To evaluate our tool, we sought to understand whether the tool could produce the intended mapping formula if the simple

Database Schema	Number of Tables	Number of Columns	Ontology	Number of Nodes	Number of Links
UTCS Department	8	32	Academic Department	62	1913
VLDB Conference	9	38	Academic Conference	27	143
DBLP Bibliography	5	27	Bibliographic Data	75	1178
OBSERVER Project	8	115	Bibliographic Data	75	1178
Country	6	18	CIA factbook	52	125

Table 3. Characteristics of Schemas and ontologies for the Experiments.

correspondences were given. We were concerned about the number of formulas presented by the tool for users to sift through. Further, we wanted to know whether the tool was still useful if the correct formula was not generated. In this case, we expected that a user could more easily debug a generated formula to reach the correct one instead of creating it from scratch. A summary of the experimental results are listed in Table 4 which shows the average size of each relational table schema in each database, the average number of candidates generated, and the average time for generating the candidates. Notice that the number of candidates is the number of semantic trees obtained by the algorithm. Also, a single edge of an semantic tree may represent the multiple edges between two nodes, collapsed using our $p; q$ abbreviation. If there are m edges in a semantic tree and each edge has n_i $i = 1, \dots, m$ original edges collapsed, then there are $\prod_i^m n_i$ original semantic trees. We show below a formula generated from such a collapsed semantic tree:

TaAssignment(courseName, studentName) :-

Course(x_1), GraduateStudent(x_2), **hasTAs;takenBy**(x_1, x_2),
workTitle($x_1, \text{courseName}$), personName($x_2, \text{studentName}$).

where, in the semantic tree, the node *Course* and the node *GraduateStudent* are connected by a single edge with label **hasTAs;takenBy**, which represents two separate edges, *hasTAs* and *takenBy*.

Database Schema	Avg. Number of Cols/per table	Avg. Number of Candidates generated	Avg. Execution time(ms)
UTCS Department	4	4	279
VLDB Conference	5	1	54
DBLP Bibliography	6	3	113
OBSERVER Project	15	2	183
Country	3	1	36

Table 4. Performance Summary for Generating Mappings from Relational Tables to Ontologies.

Table 4 indicates that MAPONTO only presents a few mapping formulas for users to examine. This is due in part to our compact representation of parallel edges between two nodes shown above. To measure the overall performance, we manually created the mapping formulas for all the 36 tables and compared them to the formulas generated by the tool. We observed that the tool produced correct formulas for 31 tables. This demonstrates that the tool is able to infer the semantics of many relational tables occurring in practice in terms of an independently developed ontology.

We were also interested in the usefulness of the tool in those cases where the formulas generated were not the intended ones. For each such formula, we compared it to the manually generated correct one, and we used a very coarse measurement to record how much effort it would take to “debug” the generated formula: the number of changes of predicate names in a formula. For example, the tool generated the following formula for the table *Student(name, office, position, email, phone, supervisor)*:

Student(X₁), emailAddress(X₁,email), personName(X₁,name), Professor(X₂), Department(X₃), head(X₃,X₂), affiliatedOf(X₃,X₁), personName(X₂, supervisor)... (1)

If the intended semantics for the above table columns is:

Student(X₁), emailAddress(X₁,email), personName(X₁,name), Professor(X₂), ResearchGroup(X₃), head(X₃,X₂), affiliatedOf(X₃,X₁), personName(X₂, supervisor)... (2)

then one can change the predicate *Department(X₃)* to *ResearchGroup(X₃)* in formula (1) instead of writing the entire formula (2) from scratch. Our experience working with the tested data sets shows that at average only about 30% predicates in the formula returned by the MAPONTO tool needed to be modified to reach the correct formula. This is a significant saving in terms of human labors.

Tables 4 indicate that execution times were not significant, since, as predicted, the search for subtrees and paths took place in a relatively small neighborhood.

We believe it is instructive to consider the various categories of problematic schemas/mappings, and the kind of future work they suggest.

(i) *Absence of tables which should be present according to er2rel.* For example, we expect the connection Person -- researchInterest --- Research to be returned for the table *AreaOfInterest(name, area)*. However, MAPONTO returned Person -<- headOf --- ResearchGroup -<- researchProject --- Research, because there was no table for the concept *Research* in the schema, and so MAPONTO treated it as a weak entity table. The elimination, during schema design, of tables that represent finite enumerations, or can be recovered by projection from tables representing total many-to-many relationships, will cause such ambiguities, and is an important open problem for now.

(ii) *Differences in Representation.* The table *Presentation(event, presenter, paper, start, end)* represents an n-ary relationship among three entities. However, in the ontology, the *Presentation* concept is a subclass of *Event* concept and has *Presenter* and *Paper* as role fillers.

(iii) *Mapping formula requiring select.* The table *European(country, gnp)* means countries which are located in Europe. From the database point of view, this selects tuples representing European countries. Currently, MAPONTO is incapable of generating formulas involving the equivalent to relational selection. This particular case is an instance of the need to express “higher-order” correspondences, such as between table/column names and ontology values. A similar example appears in [14].

(iv) *Non-standard design* One of the bibliography tables had columns for *author* and *otherAuthors* for each document. MAPONTO found a formula that was close to the desired one, with conjuncts *hasAuthor(d, author)*, *hasAuthor(d, otherAuthors)*, but not surprisingly, could not add the requirement that *otherAuthors* is really the concatenation of all but the first author.

7 Refining Mappings

Filtering Mappings by Ontology Reasoning Rich ontologies provide a new opportunity for eliminating “unreasonable” mappings. For example, if the ontology specifies that in a library once a *Book* is reserved for an event, it cannot be borrowed by a *Person*, then a candidate semantic formula *Book(X), borrow(X, Y), Person(Y), reservedFor(X, Z), Event(Z)* can be eliminated¹², since no objects *X* can satisfy it. When ontologies, including constraints such as the one about *borrowing/reservedFor*, are expressed in OWL, one can actually use OWL reasoning to detect inconsistent semantics by converting semantic trees into OWL concepts, and then testing them for incoherence with respect to the ontology. For example, the above formula can be translated into the OWL concept whose abstract syntax is *intersectionOf(Book, restriction(borrow some ValuesFrom(Person)), restriction(reservedFor some ValuesFrom(Event)))*. The algorithm for performing this translation is almost identical to `encodeTree(S,L)`, except that the recursive calls return OWL concepts C_i , which lead to conjuncts of the form *restriction(p_i , someValuesFrom(C_i))*. The ontologies we have found so far are unfortunately not sufficiently rich to demonstrate the usefulness of this idea.

Finding GAV Mappings Arguments have been made that the proper way to connect ontologies and databases for the purpose of information integration is to show how concepts and properties in the ontology can be expressed as queries over the database – the so-called GAV approach. Our techniques can be adapted for this purpose relatively easily: given a table, *emp(eid, did)*, whose semantics

¹² Probably a relationship like *contactAuthor(X, Y)* other than *borrow(X, Y)* needs to be used.

w.r.t. Figure 1 was computed to be

$$\begin{aligned} T:\text{emp}(\text{sin},\text{did}) &:- \mathcal{O}:\text{Employee}(x), \mathcal{O}:\text{hasSsn}(x,\text{sin}), \\ &\quad \mathcal{O}:\text{Department}(y), \mathcal{O}:\text{hasDeptNumber}(y,\text{did}), \\ &\quad \mathcal{O}:\text{works_for}(x,y). \end{aligned}$$

its actual semantics is expressed by the logical formula

$$\begin{aligned} \forall \text{sin}, \text{did} . T:\text{emp}(\text{sin}, \text{did}) &\Rightarrow \exists x, y. \mathcal{O}:\text{Employee}(x) \wedge \mathcal{O}:\text{hasSsn}(x, \text{sin}) \wedge \\ &\quad \mathcal{O}:\text{Department}(y) \wedge \mathcal{O}:\text{hasDeptNumber}(y, \text{did}) \wedge \mathcal{O}:\text{works_for}(x, y). \end{aligned}$$

because not all objects in the world of the ontology need be known in the database. This formula can in fact be Skolemized to eliminate the existential quantifiers to yield

$$\begin{aligned} \forall \text{sin}, \text{did} . T:\text{emp}(\text{sin}, \text{did}) &\Rightarrow \mathcal{O}:\text{Employee}(f(\text{sin}, \text{did})) \wedge \\ &\quad \mathcal{O}:\text{hasSsn}(f(\text{sin}, \text{did}), \text{sin}) \wedge \mathcal{O}:\text{Department}(g(\text{sin}, \text{did})) \wedge \\ &\quad \mathcal{O}:\text{hasDeptNumber}(g(\text{sin}, \text{did}), \text{did}) \wedge \mathcal{O}:\text{works_for}(f(\text{sin}, \text{did}), g(\text{sin}, \text{did})). \end{aligned}$$

which in turn can be used to obtain GAV rules for each of the conjuncts, such as $\forall \text{sin}, \text{did} . \mathcal{O}:\text{hasSsn}(f(\text{sin}, \text{did}), \text{sin}) \Leftarrow T:\text{emp}(\text{sin}, \text{did})$ and $\forall \text{sin}, \text{did} . \mathcal{O}:\text{works_for}(f(\text{sin}, \text{did}), g(\text{sin}, \text{did})) \Leftarrow T:\text{emp}(\text{sin}, \text{did})$. (This technique is known as the “inverse rule” for query answering using views [16].)

The problem is that different tables introduce different skolem functions, so that there seems to be no way to “join” $\mathcal{O}:\text{works_for}(f(\text{sin}, \text{did}), g(\text{sin}, \text{did}))$ with $\mathcal{O}:\text{manages}(h(\text{sin}, \text{name}, \text{did}), k(\text{sin}, \text{name}, \text{did}))$ on the first argument, say, despite the fact that the arguments correspond to employees in both case. It turns out that because $\mathcal{O}:\text{hasSsn}$ is inverse functional (*sin* is a key), $\mathcal{O}:\text{hasSsn}(f(\text{sin}, \text{did}), \text{sin})$ and $\mathcal{O}:\text{hasSsn}(h(\text{sin}, \text{name}, \text{did}), \text{sin})$ imply that $\forall \text{sin}, \text{did}, \text{name} . f(\text{sin}, \text{did}) = h(\text{sin}, \text{name}, \text{did})$, so that in fact *all* occurrence of both functions can be replaced by a simpler one, $f'(\text{sin})$, in all the GAV rules. Such a function can of course be seen as using keys to introduce object identifiers, which are missing in the relational model of the data but are in the conceptual model.

8 Conclusion and Future Work

We have proposed a tool to infer semantic mapping formulas between relational tables and ontologies starting from simple correspondences, which relies on information from the database schema (key and foreign key structure) and the ontology (cardinality restrictions, **is-a** hierarchies). Theoretically, our algorithm infers all and only the relevant semantics if a table’s schema follows ER design principles. In practice, our experience working with independently developed schemas and ontologies has shown that significant effort has been saved in specifying the LAV mapping formulas.

Numerous additional sources of knowledge, including richer ontologies, actual data stored in the tables, linguistic and semantic relationships between identifiers in tables and the ontology, can be used to refine the suggestions of MAPONTO, including providing a rank ordering for them. As in the original Clio system, more complex correspondences (e.g., from columns to sets of attribute names or class names), should also be investigated in order to generate the full range of mappings encountered in practice.

Acknowledgments: We are most grateful to Renée Miller and Yannis Velegarakis for their clarifications concerning Clio, comments on our results, and encouragement. Remaining errors are, of course, our own.

References

1. D. Calvanese, G. D. Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Data Integration in Data Warehousing. *J. of Coop. Info. Sys.*, 10(3):237–271, 2001.
2. M. Dahchour and A. Pirotte. The Semantics of Reifying n-ary Relationships as Classes. In *ICEIS'02*, pages 580–586, 2002.
3. R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In *SIGMOD'04*, pages 383–394, 2004.
4. A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening Ontologies with DOLCE. In *EKAW'02*, pages 166–181, 2002.
5. F. Goasdoue et al. Answering queries using views: A KRDB perspective for the semantic web. *ACM TOIT*, 4(3), 2004.
6. J.-L. Hainaut. *Database Reverse Engineering*. <http://citeseer.ist.psu.edu/article/hainaut98database.html>, 1998.
7. S. Handschuh, S. Staab, and R. Volz. On Deep Annotation. In *Proc. WWW'03*, 2003.
8. P. Johannesson. A method for transforming relational schemas into conceptual schemas. In *ICDE*, pages 190–201, 1994.
9. H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications. In *ISWC2004*, Nov. 2004.
10. A. Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *J. of Intelligent Info. Sys.*, 5(2):121–143, Dec 1996.
11. A. Y. Levy. Logic-Based Techniques in Data Integration. In Jack Minker (ed), *Logic Based Artificial Intelligence.*, Kluwer Publishers, 2000
12. V. M. Markowitz and J. A. Makowsky. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. *IEEE TSE*, 16(8):777–790, August 1990.
13. E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An Approach for Query Processing in Global Information Systems Based on Interoperation Across Preexisting Ontologies. In *CoopIS'96*, pages 14–25, 1996.
14. R. Miller, L. M. Haas, and M. A. Hernandez. Schema Mapping as Query Discovery. In *VLDB'00*, pages 77–88, 2000.
15. L. Popa, Y. Velegarakis, R. J. Miller, M. Hernandez, and R. Fagin. Translating Web Data. In *VLDB'02*, pages 598–609, 2002.
16. Xiaolei Qian. Query Folding. In *Proc. ICDE*, 48-55, 1996.
17. M. R. Quillian. Semantic Memory. In *Semantic Information Processing*. Marvin Minsky (editor). 227-270. The MIT Press. 1968.
18. E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10:334–350, 2001.
19. R. Ramakrishnan and M. Gehrke. *Database Management Systems (3rd ed.)*. McGraw Hill, 2002.
20. H. Wache, T. Vogege, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hubner. Ontology-Based Integration of Information - A Survey of Existing Approaches. In *IJCAI'01 Workshop. on Ontologies and Information Sharing*, 2001.