

# Accepted Manuscript

## Ontology-Based Mappings

Giansalvatore Mecca, Guillem Rull, Donatello Santoro, Ernest Teniente

PII: S0169-023X(15)00048-8  
DOI: doi: [10.1016/j.datak.2015.07.003](https://doi.org/10.1016/j.datak.2015.07.003)  
Reference: DATAK 1529

To appear in: *Data & Knowledge Engineering*



Please cite this article as: Giansalvatore Mecca, Guillem Rull, Donatello Santoro, Ernest Teniente, Ontology-Based Mappings, *Data & Knowledge Engineering* (2015), doi: [10.1016/j.datak.2015.07.003](https://doi.org/10.1016/j.datak.2015.07.003)

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

# Ontology-Based Mappings

Giansalvatore Mecca<sup>a</sup>, Guillem Rull<sup>b,1</sup>, Donatello Santoro<sup>a</sup>, Ernest Teniente<sup>c,1</sup>

<sup>a</sup>*Università della Basilicata — Potenza, Italy*

<sup>b</sup>*Universitat de Barcelona — Barcelona, Spain*

<sup>c</sup>*Universitat Politècnica de Catalunya — Barcelona, Spain*

---

## Abstract

Data translation consists of the task of moving data from a source database to a target database. This task is usually performed by developing mappings, i.e. executable transformations from the source to the target schema. However, a richer description of the target database semantics may be available in the form of an ontology. This is typically defined as a set of views over the base tables that provides a unified conceptual view of the underlying data. We investigate how the mapping process changes when such a rich conceptualization of the target database is available. We develop a translation algorithm that automatically rewrites a mapping from the source schema to the target ontology into an equivalent mapping from the source to the target databases. Then, we show how to handle this problem when an ontology is available also for the source. Differently from previous approaches, the language we use in view definitions has the full power of non-recursive Datalog with negation. In the paper, we study the implications of adopting such an expressive language. Experiments are conducted to illustrate the trade-off between expressibility of the view language and efficiency of the chase engine used to perform the data exchange.

*Keywords:* mapping, ontology, view, tuple generating dependency, equality generating dependency, disjunctive embedded dependency.

---

## 1. Introduction

Integrating data coming from disparate sources is a crucial task in many applications. An essential requirement of any data integration task is that of manipulating *mappings* between sources. Mappings are executable transformations that define how an instance of a source repository can be translated into an instance of a target repository. Traditionally, mappings are developed to exchange data between two relational database schemas [1]. A rich body of research has been devoted to the study of this subject. This includes the

---

<sup>1</sup>This work has been partially supported by the Ministerio de Ciencia y Tecnología under projects TIN2011-24747 and TIN2014-52938-C2-2-R.

development of algorithms to simplify the specification of the mapping [2], the formalization of the semantics of the translation process [3], and various notions of quality of the results [4, 5, 6].

This paper investigates how the mapping process changes in the presence of richer *ontology schemas* of the two data sources. Studying this variant of the problem is important for several reasons.

(i) First, the emergence of the Semantic Web has increased the number of data sources on top of which ontology-like descriptions are developed.

(ii) Second, ontologies play a key role in information integration since they are used to give clients a global conceptual view of the underlying data, which in turn may come from external, independent, heterogeneous, multiple information systems [7]. On the contrary, the global unified view given by the ontology is constructed independently from the representation adopted for the data stored at the sources.

(iii) Finally, many of the base transactional repositories used in complex organizations by the various processes and applications often undergo modifications during the years, and may lose their original design. The new schema can often be seen as a set of views over the original one. It is important to be able to run the existing mappings against a view over the new schema that does not change, thus keeping these modifications of the sources transparent to the users.

It is therefore important to study how the mapping process changes in this setting.

### 1.1. Contributions

In this paper, we assume that an ontology is provided for the target and, possibly, for the source data repository. The relationship between the domain concepts in this ontology schema and the data sources is given by a set of views that define the ontology constructs in terms of the logical database tables using a relational language of conjunctive queries, comparisons and negations.

We develop a number of techniques to solve this kind of *ontology-based mapping problem*. More specifically:

- we develop rewriting algorithms to automatically translate mappings over the ontology schema into mappings over the underlying databases; we first discuss the case in which an ontology schema is available for the target database only; then we extend the algorithm to the case in which an ontology schema is available both for the source and the target;
- the algorithm that rewrites a source-to-ontology mapping into a classical and executable source-to-target mapping is based on the idea of unfolding views in mapping conclusions; in our setting this unfolding is far from being straightforward; in the paper, we show that the problem is made significantly more complex by the expressibility of the view-definition language, and more precisely, by the presence of negated atoms in the body of view definitions;

- 51 • we study the implications of adopting such an expressive language; to

52 handle negation in view definitions we adopt a very expressive mapping

53 language, namely, that of *disjunctive embedded dependencies (deds)* [8].

54 Deds are mapping dependencies that may contain disjunctions in their

55 heads, and are therefore more expressive than standard embedded depen-

56 dencies (tgds and egds);
- 57 • this increased expressive power makes the data-exchange step significantly

58 more complex. As a consequence, we investigate restrictions to the view-

59 definition language that may be handled using standard embedded de-

60 pendencies, for which efficient execution strategies exist. In the paper, we

61 identify a restricted view language that still allows for a limited form of

62 negation, but represents a good compromise between expressibility and

63 complexity; we prove that under this language, our rewriting algorithm

64 always returns standard embedded dependencies;
- 65 • the classical approach to executing a source-to-target exchange consists

66 of running the given mappings using a *chase* engine [3]. We build on the

67 LLUNATIC chase engine [9, 10], and extend it to execute not only standard

68 tgds and egds, but also deds. We discuss the main technical challenges

69 related to the implementation of deds. Then, using the prototype, we

70 conduct several experiments on large databases and mapping scenarios

71 to show the trade-offs between expressibility of the view language, and

72 efficiency of the chase. To the best of our knowledge, this is the first

73 practical effort to implement execution strategies for deds, and may pave

74 the way for further studies on the subject.

75 This paper represents a significant step forward towards the goal of incorpo-

76 rating richer ontology schemas into the data translation process. Given the

77 evolution of the Semantic Web, and the increased adoption of ontologies, this

78 represents an important problem that may lead to further research directions.

79 This paper extends our prior research [11], where we first studied the prob-

80 lem of rewriting ontology-based mappings. We make several important advance-

81 ments, as follows:

- 82 (i) First, previous papers only discussed rewritings based on standard embedded

83 dependencies for a rather limited form on negation. In this paper, we extend

84 our algorithms to handle arbitrary non-recursive Datalog with negation using

85 deds, thus considerably extending the reach of our rewriting algorithm.
- 86 (ii) At the same time, we make the sufficient conditions under which the rewrit-

87 ing only contains embedded dependencies more precise, and extend the limited

88 case discussed in previous papers.
- 89 (iii) In addition, we present the first chase technique for deds, and a comprehen-

90 sive experimental evaluation based on scenarios with and without deds. As we

91 mentioned above, this is the first practical study of the scalability of the chase

92 of high-complexity dependencies, an important problem in data exchange.
- 93 (iv) Finally, we provide full proofs of all theorems (in Appendix A).

## 1.2. Outline

The paper is organized as follows. Our motivating example is given in Section 2. Section 3 recalls some basic notions and definitions. Section 4 introduces the ontology-based mapping problem. Section 5 defines disjunctive embedded dependencies which are required by the rewriting when the views that define the mapping are beyond conjunctive queries. Section 6 provides the definition of a correct rewriting. The rewriting algorithm and formal results are in Section 7. Section 8 identifies a view-definition language that is more expressive than plain conjunctive queries but such that it computes correct rewritings only in terms of embedded dependencies. The chase engine is described in Section 9. Experiments are in Section 10. We discuss related work in Section 11.

## 2. Motivating Example

Assume we have the two relational schemas below and we need to translate data from the source to the target.

Source schema: *S-WorkerGrades*(*WorkerId*, *Year*, *Grade*, *SalaryInc*)  
*S-Stats*(*WorkerId*, *WorkerName*, *MinGrade*, *MaxGrade*)

Target schema: *Employees*(*Id*, *Name*)  
*Evaluations*(*EmployeeId*, *Year*)  
*PositiveEvals*(*EmployeeId*, *Year*, *SalaryInc*)  
*Penalized*(*EmployeeId*, *Year*)  
*Warned*(*EmployeeId*, *Date*)

Both schemas rely on the same domain, which includes data about employees and the evaluations they receive during the years. The source database stores grades within the *S-WorkerGrades* table, and statistical data in the form of minimum and maximum grades of workers in table *Stats*. The target database, on the contrary, stores data about employees and their positive evaluations, but also records warnings and penalties for those employees.

Due to these different organizations, it is not evident how to define the source-to-target mapping. In particular, it is difficult to relate information stored in table *S-Stats* from the source schema to the contents of the tables *Penalized* and *Warned* in the target schema.

Suppose now that a richer ontology has been defined over the target relational schema, as shown in Figure 1. The ontology distinguishes among problematic, average, and outstanding workers, and it records whether the yearly evaluation of each worker is negative or positive, storing also the salary increase to apply to the worker for positive evaluations.

Each class and association in the ontology is defined in terms of the database tables by means of a set of views, as follows (to simplify the reading, from now on we use different fonts for ontology classes and relational tables; in addition, source tables have a *S*-prefix in their name to be distinguished from base target tables):<sup>2</sup>

<sup>2</sup>The rules we use to specify views in our example are not safe in the sense that they contain

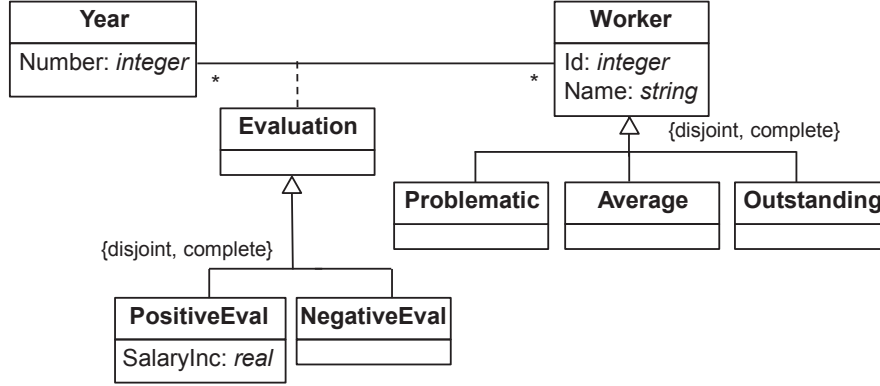


Figure 1: A Simple Target Ontology.

**View definitions for the target ontology.**

$v_1 : \text{Worker}(\text{id}, \text{name}) \Leftarrow \text{Employees}(\text{id}, \text{name})$   
 $v_2 : \text{Evaluation}(\text{employeeId}, \text{year}) \Leftarrow \text{Evaluations}(\text{employeeId}, \text{year})$   
 $v_3 : \text{PositiveEval}(\text{employeeId}, \text{year}, \text{salaryInc}) \Leftarrow \text{Evaluation}(\text{employeeId}, \text{year}),$   
 $\quad \text{PositiveEvals}(\text{employeeId}, \text{year}, \text{salaryInc})$   
 $v_4 : \text{NegativeEval}(\text{employeeId}, \text{year}) \Leftarrow \text{Evaluation}(\text{employeeId}, \text{year}),$   
 $\quad \neg \text{PositiveEval}(\text{employeeId}, \text{year}, \text{sync})$   
 $v_5 : \text{Problematic}(\text{id}, \text{name}) \Leftarrow \text{Worker}(\text{id}, \text{name}), \text{Penalized}(\text{id}, \text{year})$   
 $v_6 : \text{Problematic}(\text{id}, \text{name}) \Leftarrow \text{Worker}(\text{id}, \text{name}), \neg \text{PositiveEval}(\text{id}, \text{year}, \text{sync})$   
 $v_7 : \text{Outstanding}(\text{id}, \text{name}) \Leftarrow \text{Worker}(\text{id}, \text{name}), \neg \text{NegativeEval}(\text{id}, \text{year}),$   
 $\quad \neg \text{Warned}(\text{id}, \text{date})$   
 $v_8 : \text{Average}(\text{id}, \text{name}) \Leftarrow \text{Worker}(\text{id}, \text{name}), \neg \text{Outstanding}(\text{id}, \text{name}),$   
 $\quad \neg \text{Problematic}(\text{id}, \text{name})$

128 The process of defining semantic abstractions over databases can bring benefits  
 129 to data architects only as long as the view-definition language is expressive  
 130 enough. To this end, the view-definition language adopted in this paper goes  
 131 far beyond plain conjunctive queries, and has the full power of non-recursive  
 132 Datalog [12] with negation. In fact:

133 (i) we allow for negated atoms in view definitions; these may either correspond  
 134 to negated base tables, as happens in view  $v_7$  (table *Warned*), or even to negated  
 135 views, as in  $v_4$  (view *PositiveEval*),  $v_6$  (*PositiveEval*),  $v_7$  (*NegativeEval*) and  $v_8$   
 136 (*Outstanding* and *Problematic*);

137 (ii) views can be defined as unions of queries; in our example, *Problematic*  
 138 workers are the ones that either have been penalized or have received no positive

---

variables appearing in negative literals that do not appear in a positive one. This is done for the sake of readability since it is well-known that there is an equivalent safe rewriting for such rules.

139 evaluations at all.

140 The semantics of this ontology is closer to the way the information is stored  
141 in the source schema than the one provided by the physical target tables (notice  
142 how the ontology hides tables *Penalized* and *Warned*). Therefore, the mapping  
143 designer will find it easier to define a mapping from the source schema to the  
144 target ontology. For instance, s/he could realize that the classification of work-  
145 ers as **Average**, **Outstanding** and **Problematic** in the ontology corresponds to a  
146 ranking of workers based on their grades in the source schema. In this way,  
147 employees with grades consistently above 9 (out of 10) are outstanding, those  
148 always graded less than 4 are considered to be problematic, and the rest are  
149 average.

150 As is common [3], we use *tuple generating dependencies (tgds)* and *equality-*  
151 *generating dependencies (egds)* [8] to express the mapping. In our case, the  
152 translation of source tuples into the **Average**, **Outstanding** and **Problematic** target  
153 concepts can be expressed by using the following tgds with comparison atoms:

$$\begin{aligned}
 m_0 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad maxgr > 4, mingr < 9 \rightarrow \text{Average}(id, name) \\
 m_1 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad mingr \geq 9 \rightarrow \text{Outstanding}(id, name) \\
 m_2 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad maxgr \leq 4 \rightarrow \text{Problematic}(id, name) \\
 m_3 : & \forall id, yr, gr, sinc : \\
 & S\text{-}WorkerGrades(id, yr, gr, sinc), gr \geq 5 \rightarrow \text{PositiveEval}(id, yr, sinc) \\
 m_4 : & \forall id, yr, gr, sinc : \\
 & S\text{-}WorkerGrades(id, yr, gr, sinc), gr < 5 \rightarrow \text{NegativeEval}(id, yr)
 \end{aligned}$$

154 Intuitively, tgd  $m_0$  specifies that, for each pair of tuples in the source tables  
155 *S-WorkerGrades* and *S-Stats* that have the same value for the *id* attribute and  
156 have a *maxgrade* attribute greater than 4 and a *mingrade* attribute lower than  
157 9, there should be a worker ranked as average in the ontology. Similarly for  $m_1$   
158 and  $m_2$  for **Outstanding** and **Problematic**, respectively.

159 Mappings  $m_3$  and  $m_4$  relate the workers' evaluation data in *S-WorkerGrades*  
160 to the instances **PositiveEval** and **NegativeEval**, respectively, using the grade to  
161 discriminate between the two subclasses of **Evaluation**.

162 Notice that mappings  $m_0, m_1$  and  $m_2$  do not completely encode the seman-  
163 tics of the desired transformation. In fact, an important part of the mapping  
164 process is to generate *solutions*, i.e. instances of the target that comply with the  
165 integrity constraints imposed over the database. To do this, it is necessary to  
166 incorporate the specification of these constraints into the mapping itself. This  
167 can be done easily using additional dependencies. The mapping literature [2]  
168 usually treats target dependencies in a different way. In fact, it is custom-  
169 ary to embed foreign-key constraints into the source-to-target tgds that express  
170 the mapping. In contrast, egds require special care [6], and therefore must be  
171 expressed as separate dependencies.

Mapping  $e_0$  below is an example of an egd used to express the key constraint on **Worker**: it states that whenever two workers have the same **id**, their names must also be the same:

$$e_0 : \forall id, name_1, name_2 : \text{Worker}(id, name_1), \text{Worker}(id, name_2) \rightarrow name_1 = name_2$$

We want to emphasize the benefits of designing the mappings wrt the richer target ontology rather than wrt to the base tables. By taking advantage of the semantics of the ontology, the mapping designer does not need to care about the physical structure of the data in the target schema. As an example, s/he does not need to explicitly state in  $m_0, m_1, m_2$  that average, outstanding, and problematic workers are also workers, nor that a positive or negative evaluation is also an evaluation in  $m_3, m_4$ . The class-subclass relationships are encoded within the ontology schema, and we expect their semantics to carry on into the mappings.

However, this increased flexibility comes at a cost. For example, mappings  $m_0$  to  $m_4$  above are not directly executable, since they refer to virtual entities — the constructs in the ontology schema — and not to the actual tables in the target. We therefore need to devise a way to translate such a *source-to-ontology* mapping into a classical source-to-target mapping, in order to execute the latter and move data from the source to the target database.

The main technical problem addressed in this paper can therefore be stated as follows: given a source-to-ontology mapping, a target ontology schema, and the views defining this ontology schema in terms of the underlying database tables, we want to obtain the corresponding executable source-to-target mapping.

### 3. Preliminary Notions

In this paper, we deal with mapping scenarios that involve two levels: the ontology and the database level. This section first introduces the basic concepts of these two levels, and then elaborates on the language of dependencies used to express mapping scenarios.

#### 3.1. Databases and Ontologies

**Databases** We focus on the relational setting. A *schema*  $\mathbf{S}$  is a set of relation symbols  $\{R_1, \dots, R_n\}$ , each with an associated relation schema  $R(A_1, \dots, A_m)$ . Given schemas  $\mathbf{S}, \mathbf{T}$  with disjoint relations symbols,  $\langle \mathbf{S}, \mathbf{T} \rangle$  denotes the schema corresponding to the union of  $\mathbf{S}$  and  $\mathbf{T}$ . An *instance* of a schema is a set of tuples in the form  $R(v_1, \dots, v_m)$ , where each  $v_i$  denotes either a constant, typically denoted by  $a, b, c, \dots$ , or a *labeled null*, denoted by  $N_1, N_2, \dots$ . Constants and labeled nulls form two disjoint sets. Given instances  $I$  and  $J$ , a homomorphism  $h : I \rightarrow J$  is a mapping from  $\text{dom}(I)$  to  $\text{dom}(J)$  such that for every  $c \in \text{CONST}$ ,  $h(c) = c$ , and for all tuples  $t = R(v_1, \dots, v_n)$  in  $I$ , it is the case that  $h(t) = R(h(v_1), \dots, h(v_n))$  belongs to  $J$ . Homomorphisms immediately



211 extend to formulas, since atoms in formulas can be seen as tuples whose values  
212 correspond to variables.

213 **Ontologies** In this paper, we focus on ontologies that deal with static aspects.  
214 In particular, we consider ontologies that consist of a taxonomy of entity types  
215 (which may have attributes), a taxonomy of relationship types (defined among  
216 entity types), and a set of integrity constraints (which affect the state of the  
217 domain). The integrity constraints are expressed by means of *dependencies* (see  
218 Section 3.2).

**Views** To bridge the gap between the ontology schema and the underlying  
database, we assume that a set of GAV views (Global-As-View) is given for each  
entity and relationship type, which defines this type in terms of the underlying  
database. A *view*  $V$  is a derived relation defined over a schema  $S$ . The view  
definition for  $V$  over  $S$  is a non-recursive rule of the form:

$$v : V(\bar{x}) \Leftarrow R_1(\bar{x}_1), \dots, R_p(\bar{x}_p), \neg R_{p+1}(\bar{x}_{p+1}), \dots, \neg R_{p+g}(\bar{x}_{p+g})$$

219 with  $p \geq 1$  and  $g \geq 0$ , where the variables in  $\bar{x}$  are taken from  $\bar{x}_1, \dots, \bar{x}_p$ .  
220 Atoms in a view definition can be either base or derived. An atom  $V(\bar{x})$  is a  
221 *derived atom* if  $V$  denotes a view; otherwise it is a *base atom*. A view definition  
222 specifies how the extension of the view is computed from a given instance of  
223 the underlying schema, that is, given a homomorphism  $h$  from the definition of  
224  $V$  to an instance  $I$ ,  $h(V(\bar{x}))$  belongs to the extension of  $V$  iff  $h(R_1(\bar{x})) \wedge \dots \wedge$   
225  $\neg h(R_{p+g}(\bar{x}_{p+g}))$  is true on  $I$ .

### 226 3.2. Dependencies and Mapping Scenarios

227 **Dependencies** A *tuple-generating dependency* (*tg**d*) over  $\mathbf{S}$  is a formula of the  
228 form  $\forall \bar{x}, \bar{z} (\phi(\bar{x}, \bar{z}) \rightarrow \exists \bar{y} \psi(\bar{x}, \bar{y}))$ , where  $\phi(\bar{x}, \bar{z})$  and  $\psi(\bar{x}, \bar{y})$  are conjunctions of  
229 atoms. We allow two kinds of atoms in the premise: (a) relational atoms over  
230  $\mathbf{S}$ ; (b) comparison atoms of the form  $v \text{ op } c$ , where *op* is a comparison operator  
231 ( $=, >, <, \geq, \leq$ ),  $v$  is a variable that also appears as part of a relational atom,  
232 and  $c$  is a constant. Only relational atoms are allowed in the conclusion.

233 An *equality generating dependency* (*egd*) over  $\mathbf{S}$  is a formula of the form  
234  $\forall \bar{x} (\phi(\bar{x}) \rightarrow x_i = x_j)$  where  $\phi(\bar{x})$  is a conjunction of relational atoms over  $\mathbf{S}$   
235 and comparison atoms as defined above, and  $x_i$  and  $x_j$  occur in  $\bar{x}$ . A *denial*  
236 *constraint* is a special form of *egd* of the form  $\forall \bar{x} (\phi(\bar{x}) \rightarrow \perp)$ , in which the  
237 conclusion only contains the  $\perp$  atom, which cannot be made true. Tgds and  
238 egds [8] form the language of *embedded dependencies*.

239 **Mapping Scenarios** A *mapping scenario* [3],  $\mathcal{M} = \{\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T\}$ , is a  
240 quadruple consisting of:

- 241 • a source schema  $\mathbf{S}$ ;
- 242 • a target schema  $\mathbf{T}$ ;
- 243 • a set of *source-to-target* (*s-t*) *tgds*  $\Sigma_{ST}$ , i.e. tgds such that the premise is  
244 a formula over  $\mathbf{S}$  and the conclusion a formula over  $\mathbf{T}$ ;

- a set  $\Sigma_T$  of *target tgds* — tgds over  $\mathbf{T}$  — and *target egds* — egds over  $\mathbf{T}$ .

Given a source instance  $I$ , a solution for  $I$  under  $\mathcal{M}$  is a target instance  $J$  such that  $I$  and  $J$  satisfy  $\Sigma_{ST}$ , and  $J$  satisfies  $\Sigma_T$ . A solution  $J$  for  $I$  and  $\mathcal{M}$  is called a *universal solution* if, for all other solutions  $J'$  for  $I$  and  $\mathcal{M}$ , there is a homomorphism from  $J$  to  $J'$ . The chase is a well-known algorithm for computing universal solutions [3]. We denote by  $\text{Sol}(\mathcal{M}, I)$  the set of solutions for  $\mathcal{M}$  and  $I$ , and by  $\text{USol}(\mathcal{M}, I)$  the set of universal solutions for  $\mathcal{M}$  and  $I$ .

#### 4. The Ontology-Based Mapping Problem

The goal of this section is to introduce our mapping problem. Let us first assume that an ontology schema is only available for the target database (case a). Then, we discuss how things can be extended to handle a source ontology as well (case b).

##### 4.1. Case a: Source-to-Ontology Mappings

The inputs to our source-to-ontology mapping problem are:

1. a source relational schema,  $\mathbf{S}$ , and a target relational schema  $\mathbf{T}$ ;
2. a target ontology schema,  $\mathbf{V}$ , defined by means of a set of view definitions,  $\Upsilon_{TV}$ , over  $\mathbf{T}$ . View definitions may involve negations over derived atoms, as discussed in Section 3;
3. a set of target constraints,  $\Sigma_V$ , i.e. target egds to encode key constraints and functional dependencies over the ontology schema;
4. finally, a source-to-ontology mapping,  $\Sigma_{SV}$ , defined as a set of s-t tgds over  $\mathbf{S}$  and  $\mathbf{V}$ .

Based on these, our intention is to rewrite the dependencies in  $\Sigma_{SV} \cup \Sigma_V$  as a new set of source-to-target dependencies  $\Sigma_{ST} \cup \Sigma_T$ , from the source to the target database. The process is illustrated in Figure 2a, where solid lines refer to inputs, and dashed lines to outputs produced by the rewriting.

##### 4.2. Case b: Ontology-to-Ontology Mappings

The following sections are devoted to the development of the mapping rewriting algorithm. Before we turn to that, let us discuss what happens when also an ontology schema over the source is given, as shown in 2b. In this case, we assume that in addition to the target-ontology view-definitions,  $\Upsilon_V$ , view definitions for the source ontology schema,  $\Upsilon_{V'}$ , are also given, with the respective egds. We also assume that the mapping,  $\Sigma_{V'V}$ , is designed between the two ontologies.

It can be seen that this case can be reduced to the one above. We can see the problem as the composition of two steps:

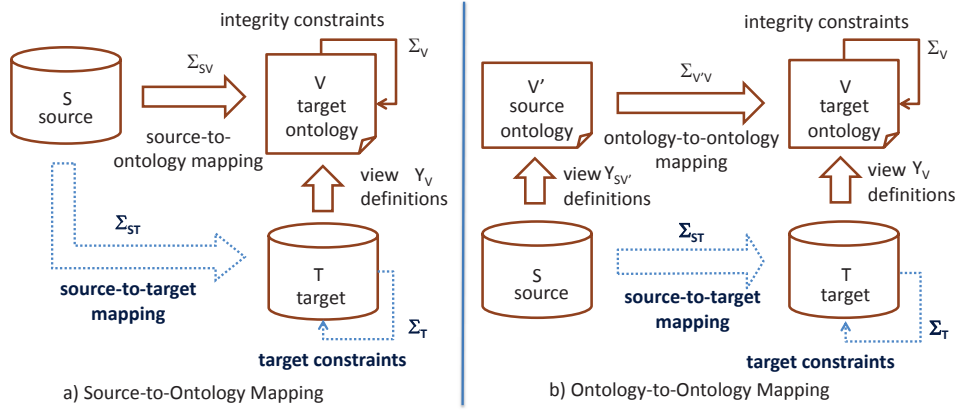


Figure 2: Ontology Mapping Scenarios.

- (i) applying the source view definitions in  $\Upsilon_{V'}$  to the source instance,  $I$ , to materialize the extent of the source ontology,  $\Upsilon_{V'}(I)$ ;
- (b) consider this materialized instance as a new source database, and solve the source-to-ontology mapping problem as in Figure 2a.

In light of this, in the following we concentrate on the scenario in Figure 2a only.

## 5. Disjunctive Embedded Dependencies

Mappings with views have been addressed in previous papers (e.g. [5, 4]). As is obvious, the complexity of the problem depends quite a lot on the expressibility of the view-definition language allowed in our scenarios. Previous works have made almost exclusive reference to views defined using the language of conjunctive queries. In this case, the rewriting consists of an application of the standard view unfolding algorithm [13].

To give an example, consider mapping  $m_3$  (from now on, we omit universal quantifiers), and recall the definition of views **PositiveEval**, and **Evaluation**:

$$\begin{aligned} m_3 : & S\text{-WorkerGrades}(id, yr, gr, sinc), gr \geq 5 \rightarrow \text{PositiveEval}(id, yr, sinc) \\ v_2 : & \text{Evaluation}(\text{employeeId}, \text{year}) \Leftarrow \text{Evaluations}(\text{employeeId}, \text{year}) \\ v_3 : & \text{PositiveEval}(\text{employeeId}, \text{year}, sinc) \Leftarrow \text{Evaluation}(\text{employeeId}, \text{year}), \\ & \text{PositiveEvals}(\text{employeeId}, \text{year}, sinc) \end{aligned}$$

Standard view unfolding replaces the view symbols of  $\text{tgdc}$  conclusions by their definitions, while appropriately renaming the variables. In our example, this yields the following s-t  $\text{tgdc}$ :

$$\begin{aligned} m'_3 : & S\text{-WorkerGrades}(id, yr, gr, sinc), gr \geq 5 \rightarrow \text{Evaluations}(id, yr), \\ & \text{PositiveEvals}(id, year, sinc) \end{aligned}$$

However, the main purpose of having a semantic description of the target database stands in its richer nature with respect to the power of the pure

selection-projection-join paradigm. In this paper we allow for a more expressive language than conjunctive queries, i.e. non-recursive Datalog with negation.

It is known [14] that the language of embedded dependencies (tgds and egds) is closed wrt unfolding conjunctive views, i.e. the result of unfolding a set of conjunctive view definitions within a set of tgds and egds is still a set of tgds and egds. A natural question is if this is also true for our more expressive view-definition language. Unfortunately, we can provide a negative answer to this question.

**Theorem 1.** *There exists a source-to-ontology mapping scenario  $\mathcal{M}_{SV} = \{\mathcal{S}, \mathcal{V}, \Sigma_{SV}, \Sigma_V\}$  with view definition  $\Upsilon_V$ , and an instance  $I$ , such that  $\mathcal{M}_{SV}$  and  $I$  admit a universal solution  $J_V \in \text{USol}(\mathcal{M}_{SV}, I)$ , and there exists no source-to-target scenario  $\mathcal{M}_{ST}$  composed of embedded dependencies (tgds and egds) such that  $\mathcal{M}_{ST}$  and  $I$  admit a solution  $J_T$ , and  $J_V = \Upsilon(J_T)$ .*

The proof of the theorem is in Appendix A. Regardless of the technical details, it is quite easy to get the intuition that stands behind this negative result: in essence, we are doomed to fail in some cases because of the limited expressive power of our mapping language. In essence, we are trying to capture the semantics of a view-definition language that allows for non-recursive negation, by means of a mapping language based on embedded dependencies, that does not use negation.

This justifies two important choices wrt the algorithm:

(i) To start, we follow a best-effort approach. We design an algorithm that is sound, i.e. given  $\mathcal{M}_{SV}$ , it generates a rewritten source-to-target scenario  $\mathcal{M}_{ST}$  such that, whenever  $\mathcal{M}_{ST}$  admits a universal solution  $J_T$ , then also the original source-to-ontology  $\mathcal{M}_{SV}$  admits universal solutions on  $I$ , and it is the case that  $\Upsilon_V(J_T)$  is a solution for  $\mathcal{M}_{SV}$  and  $I$ . In other terms, we give up completeness, and say nothing about the cases in which  $\mathcal{M}_{ST}$  fails. This notion will be made more precise in the following.

(ii) To better simulate the effects of negation in view definitions, we choose a very expressive mapping language, i.e. we extend the language of embedded dependencies (tgds and egds), by introducing disjunctions in conclusions. This gives us the more expressive mapping language of *disjunctive embedded dependencies (deds)*, that we use as a target language for our rewritings, formalized as follows.

**Definition 1 (Ded).** A *disjunctive embedded dependency (ded)* is a first-order formula of the form:

$$\forall \bar{x}, \bar{z} (\varphi(\bar{x}, \bar{z}) \rightarrow \bigvee_{l=1}^n (\exists \bar{y}_l \psi_l(\bar{x}, \bar{y}_l)))$$

where  $\varphi(\bar{x}, \bar{z})$  and each  $\psi_l(\bar{x}, \bar{y}_l)$  are conjunctions of atoms. Atoms in each conjunct  $\psi_l(\bar{x}, \bar{y}_l)$  may be either relational atoms, or comparison atoms of the form  $(x_i = x_j)$ , or the special *unsatisfiable* atom  $\perp$ .

338 A ded is called a *source-to-target ded* if  $\varphi(\bar{x}, \bar{z})$  is a conjunction of relational  
 339 atoms over  $\mathbf{S}$ , and each  $\psi_l(\bar{x}, \bar{y}_l)$  is a conjunction of relational atoms over  $\mathbf{T}$ . It  
 340 is called a *target ded* if  $\varphi(\bar{x}, \bar{z})$  is a conjunction of relational atoms over  $\mathbf{T}$ , and  
 341 each  $\psi_l(\bar{x}, \bar{y}_l)$  is either a comparison atom, or a conjunction of relational atoms  
 342 over  $\mathbf{T}$ , or the unsatisfiable atom.

In essence, the conclusion of a ded is the disjunction of various conjunctions,  
 as in the following examples, where  $S_i$  are source symbols, and  $T_j$  are target  
 symbols:

$$\begin{aligned} m_{d_1} &: \forall x : S_1(x) \rightarrow (\exists y : T_1(x, y)) \vee T_2(x, x) \\ m_{d_2} &: \forall x, y : S_2(x, y) \rightarrow T_3(x, y) \vee (\exists z : T_3(x, z), T_4(z, y)) \\ m_{d_3} &: \forall x, y, z, y', z' : T_1(x, y, z), T_1(x, y', z') \rightarrow (y = y') \vee (z = z') \\ m_{d_4} &: \forall x, y, z, y', z' : T_1(x, y, z) \rightarrow (y = z) \vee T_3(x, y) \\ m_{d_5} &: \forall x, y, z, y', z' : T_1(x, y, z), T_1(x, y', z') \rightarrow \perp \end{aligned}$$

343 Here,  $m_{d_1}$  and  $m_{d_2}$  are source-to-target deds, while  $m_{d_3}$ ,  $m_{d_4}$  and  $m_{d_5}$  are target  
 344 deds. The semantics is easily explained:  $m_{d_1}$  is satisfied by instances  $I, J$  of  $\mathbf{S}$ ,  
 345  $\mathbf{T}$  if, whenever there exists in  $I$  a tuple of the form  $S_1(c)$ , where  $c$  is a constant,  
 346 then  $J$  either contains a tuple of the form  $T_1(c, v)$  (where  $v$  is a constant or a  
 347 labeled null), or it contains a tuple of the form  $T_2(c, c)$ . Similarly for  $m_{d_2}$ .

348 Based on this, it is easy to see that ded  $m_{d_3}$  states that table  $T_1$  is such  
 349 that, for any pair of tuples, whenever the first attributes are equal, then either  
 350 the second ones, or the third ones must be equal too. In this respect, this is a  
 351 generalization of an egd. It is also interesting to note that deds may freely mix  
 352 equalities and relational atoms in their conclusions, as happens with  $m_{d_4}$ .

353 Ded  $m_{d_5}$  states what is called a *denial constraint*: since its conclusion only  
 354 contains the unsatisfiable atom, then it will fail whenever the premise is satisfied,  
 355 since there is no way to satisfy the constraint. It is a way to state failure con-  
 356 ditions for the mappings, i.e. configurations of the source and target instances  
 357 for which there is no solution.

358 Clearly the definition of deds contains, for  $l = 1$ , that of the classical embed-  
 359 dedded dependencies. A *mapping scenario with deds* is a quadruple  $\mathcal{M}^{ded} =$   
 360  $\{\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T\}$  where  $\Sigma_{ST}$  is a set of *source-to-target deds* and  $\Sigma_T$  is a set of  
 361 *target deds*.

There are a few important differences between ordinary mapping scenarios  
 with embedded dependencies, and their counterpart with deds. Recall from  
 Section 3 that the semantics of ordinary mapping scenarios is centered around  
 the notion of a *universal solution*. Given a scenario  $\mathcal{M}^{emb}$  and a source instance  
 $I$ , in most cases there are countably many solutions, i.e. target instances that  
 satisfy the dependencies. Consider for example:

$$m_1 : \forall x : S_1(x) \rightarrow \exists y : T_1(x, y)$$

Given  $I = \{S_1(a)\}$ , all of the following are solutions for  $m_1$  (in the following,  
 $a, b, c, \dots$  are constants and  $N_i$  denotes a labeled null, i.e. a null value with an

explicit label introduced to satisfy existential quantifiers):

$$\begin{array}{ll} J_1 = \{T_1(a, N)\} & J_3 = \{T_1(a, b), T_1(a, N)\} \\ J_2 = \{T_1(a, b)\} & J_4 = \{T_1(a, b), T_2(b, c)\} \end{array}$$

A solution for  $\mathcal{M}^{emb}$  and  $I$  is called a *universal solution* if it has a homomorphism in every other solution for  $\mathcal{M}^{emb}$  and  $I$ . Universal solutions are considered as “good” solutions, preferable to non universal ones. The intuition behind the formal definition is that a universal solution does not introduce any unnecessary and unjustified information within the target. In fact, any unjustified tuples would not be mappable via homomorphisms in every other solution. In our example, only  $J_1$  is universal; every other solution in the example contains extra information that is not strictly necessary to enforce the *tg*d, either in the form of constants in place of nulls, or extra tuples.

As soon as we introduce *ded*s, the theoretical framework changes quite significantly. Deutsch and others have shown [15] that the definition of a universal solution is no longer sufficient for *ded*-based scenarios, and that the more appropriate notion of *universal model set* is needed.

**Definition 2 (Universal Model Set).** Given an instance  $I$  under a scenario  $\mathcal{M}^{ded}$ , a *universal model set* is a set of target instances  $\mathbf{J} = \{J_0, \dots, J_n\}$  such that:

- every  $J_i \in \mathbf{J}$  is a solution form  $\mathcal{M}^{ded}$ ;
- for every other solution  $J'$ , there exists a  $J_i \in \mathbf{J}$  such that there is a homomorphism from  $J_i$  to  $J'$ .

It is not difficult to understand why a set of different solutions is needed. Consider our *ded*  $m_{d_1}$  above. On source instance  $I = \{S_1(a)\}$ , it has two completely different solutions, namely  $J_1 = \{T_1(a, N)\}$ ,  $J_2 = \{T_2(a, a)\}$ . Neither is universal in the ordinary sense, since they cannot be mapped into one another; on the contrary, both contribute to describe the “good” ways to satisfy  $m_{d_1}$ .

In the following, we introduce our rewriting algorithm with *ded*s. Before turning to it, it is important to emphasize another crucial difference wrt standard embedded dependency in terms of the complexity of generating solutions. The chase [3] is a well known, polynomial-time procedure to generate universal solutions for standard *tg*ds and *eg*ds. It is possible, as we discuss in the following sections, to extend it to generate universal model sets for *ded*s, but at a price in terms of complexity. Universal model sets, in fact, are usually of exponential size wrt to the size of the source instance,  $I$ .

To see this, consider a simple example composed of *ded*  $m_{d_1}$  above:

$$m_{d_1} : \forall x : S_1(x) \rightarrow (\exists y : T_1(x, y)) \vee T_2(x, x)$$

Given  $I = \{S_1(a), S_1(b), S_1(c)\}$ , the universal model set for the *ded* contains eight different solutions, each one corresponding to one way to choose among

the branches in the conclusions of  $m_{d_1}$  for a tuple in  $S_1$ :

$$\mathbf{J} = \left\{ \begin{array}{ll} \{T_1(a, N_1), T_1(b, N_2), T_1(c, N_3)\}, & \{T_1(a, N_1), T_1(b, N_2), T_2(c, c)\} \\ \{T_1(a, N_1), T_2(b, b), T_1(c, N_3)\}, & \{T_1(a, N_1), T_2(b, b), T_2(c, c)\} \\ \{T_2(a, a), T_1(b, N_2), T_1(c, N_3)\}, & \{T_2(a, a), T_1(b, N_2), T_2(c, c)\} \\ \{T_2(a, a), T_2(b, b), T_1(c, N_3)\}, & \{T_2(a, a), T_2(b, b), T_2(c, c)\} \end{array} \right\}$$

394 In the general case, for source instances of size  $n$  we may have universal model  
 395 sets of  $O(k^n)$ , where  $k$  depends on the number of disjunctions in ded conclusions.  
 396 Therefore, one of the technical challenges posed by this problem is to tame this  
 397 exponential complexity.

## 398 6. Correctness

399 We need to introduce a few preliminary notions. A crucial requirement  
 400 about our rewriting algorithm is that the result of executing the source-to-target  
 401 mapping is “the same” as the one that we would obtain if the source-to-ontology  
 402 mapping were to be executed. Intuitively, we mean that a solution produced  
 403 by the source-to-target mapping induces a solution for the source-to-ontology  
 404 mapping when applying the view definitions.

405 To be more precise, consider the source-to-ontology mapping scenario:  $\mathcal{M}_{SV}$   
 406  $= \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$ . For each source instance  $I$ , assume there exists a solution  
 407  $J_V$  for  $I$  and  $\mathcal{M}_{SV}$  that complies with the view definitions in  $\Sigma_V$  (i.e. there  
 408 exists an instance  $J_T$  of schema  $\mathbf{T}$  such that  $J_V = \Upsilon_V(J_T)$ ). Figure 3a and 3b  
 409 show one example of  $I$  and  $J_V$ .

<b>a. Source instance <math>I</math></b>		
<i>S-WorkerGrades</i> (1, 2012, 7, 100)	<i>S-Stats</i> (1, John, 7, 8)	
<i>S-WorkerGrades</i> (1, 2013, 8, 200)		
<b>b. Ontology instance <math>J_{T'}</math></b>		
Average(1, John)	Worker(1, John)	
Evaluation(1, 2012)	Evaluation(1, 2013)	
PositiveEval(1, 2012, 100)	PositiveEval(1, 2013, 200)	
Year(2012)	Year(2013)	
<b>c. Target instance <math>J_T</math></b>		
<i>Employees</i> (1, John)	<i>Evaluations</i> (1, 2012)	<i>Evaluations</i> (1, 2013)
<i>PositiveEvals</i> (1, 2012, 100)	<i>PositiveEvals</i> (1, 2013, 200)	<i>Warned</i> (1, $N_1$ )

Figure 3: Source, ontology, and target instances.

410 We compute our rewriting, and obtain a new source-to-target scenario:  
 411  $\mathcal{M}_{ST} = \{\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T\}$ , where we assume that  $\Sigma_{ST}$  and  $\Sigma_T$  are sets of ded.  
 412 We may run  $\mathcal{M}_{ST}$  on  $I$  to obtain solutions under the form of target instances.  
 413 To any target instance  $J_T$  of this kind, we may apply the view definitions in  $\Upsilon_V$   
 414 in order to obtain an instance of  $\mathbf{V}$ ,  $J_V = \Upsilon_V(J_T)$ .



Our first intuition about the correctness of the algorithm is that the rewritten source-to-target scenario,  $\mathcal{M}_{ST}$ , should generate solutions, i.e. target instances that are guaranteed to generate views that, in turn, are solutions for the original source-to-ontology scenario,  $\mathcal{M}_{SV}$ . More precisely:

**Definition 3 (Correct Rewriting).** Given a source-to-ontology scenario  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$  with view definitions  $\Upsilon_V$ , we say that the source-to-target rewritten scenario  $\mathcal{M}_{ST} = \{\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T\}$  with deds is a *correct rewriting* of  $\mathcal{M}_{SV}$  if, for each instance  $I$  of the source database, whenever a universal model set  $\mathbf{J} = \{J_0, \dots, J_n\}$  for  $I$  and  $\mathcal{M}_{ST}$  exists, then for each  $J_i \in \mathbf{J}$ ,  $\Upsilon_V(J_i)$  is also a solution for  $I$  and the original scenario  $\mathcal{M}_{SV}$ .

The meaning of this definition is illustrated in Figure 4.

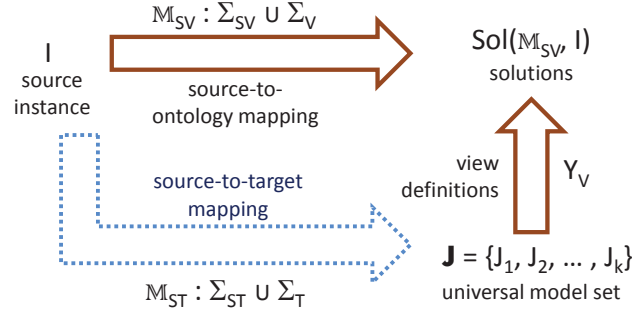


Figure 4: Correctness Diagram.

Figure 3c reports a correct target solution for  $I$  ( $N_1$  is a labeled null). Note that  $\Sigma_V(J_T)$  is exactly the ontology instance  $J_{T'}$  in Figure 3b. A different font is used for entity and relationship types in the ontology instance.

## 7. The Rewriting Algorithm

In the following, we always assume that the input mapping captures all of the semantics from the ontology level. This means that all referential constraints implicit in the ontology (i.e. ontology tgds) have to be made explicit and properly encoded into the mapping dependencies [16]. In particular, whenever a relational atom  $V(\bar{x})$  appears in the conclusion of a mapping dependency  $m$  and there is an ontology tgd  $e : V(\bar{x}) \rightarrow \psi(\bar{x})$ , we replace  $V(\bar{x})$  by  $\psi(\bar{x})$  in  $m$ . We restrict the textual integrity constraints to be key constraints and functional dependencies, and assume they are expressed as logical dependencies (i.e. egds) over the views (an automatic OCL-to-logic translation is proposed in [17]). Figure 5 shows the complete set of mapping dependencies  $\Sigma_{SV}$  for our running example.

Our algorithm generates:

- (a) a new set of source-to-target tgds,  $\Sigma_{ST}$ ;



$$\begin{aligned}
 m_0 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & \quad S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad maxgr > 4, mingr < 9 \rightarrow \text{Average}(id, name), \text{Worker}(id, name) \\
 m_1 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & \quad S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad mingr \geq 9 \rightarrow \text{Outstanding}(id, name), \text{Worker}(id, name) \\
 m_2 : & \forall id, yr, gr, sinc, name, maxgr, mingr : \\
 & \quad S\text{-}WorkerGrades(id, yr, gr, sinc), S\text{-}Stats(id, name, mingr, maxgr), \\
 & \quad maxgr \leq 4 \rightarrow \text{Problematic}(id, name), \text{Worker}(id, name) \\
 m_3 : & \forall id, yr, gr, sinc : \\
 & \quad S\text{-}WorkerGrades(id, yr, gr, sinc), gr \geq 5 \rightarrow \exists name : \text{PositiveEval}(id, yr, sinc), \\
 & \quad \text{Evaluation}(id, yr), \text{Worker}(id, name), \text{Year}(yr) \\
 m_4 : & \forall id, yr, gr, sinc : \\
 & \quad S\text{-}WorkerGrades(id, yr, gr, sinc), gr < 5 \rightarrow \exists name : \text{NegativeEval}(id, yr), \\
 & \quad \text{Evaluation}(id, yr), \text{Worker}(id, name), \text{Year}(yr)
 \end{aligned}$$

Figure 5: Source-to-ontology mapping.

- (b) a set of target dependencies,  $\Sigma_T$ . This latter set will contain:
- (b1) a set of target deds that model egds over the ontology schema. However, it may also incorporate other constraints that were not in the input. More precisely:
  - (b2) a set of target deds, i.e. deds defined over the symbols in the target only;
  - (b3) a set of denial constraints.

Denial constraints are crucial in our approach. Recall from Section 3 that a denial constraint is a dependency of the form  $\forall \bar{x}(\varphi(\bar{x}) \rightarrow \perp)$ . We use these to express the fact that some tuple configurations in the target are not compatible with the view definitions, and therefore should cause a failure in the mapping process. In other words, we are expressing part of the semantics of negations that comes with view definitions, in the form of failures of the data exchange process. This prevents our algorithm from being complete, as stated in Theorem 1, but guarantees that it is sound.

Given our input source-to-ontology mapping scenario,  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$ , our approach is to progressively rewrite dependencies in  $\Sigma_{SV}$  and  $\Sigma_V$  in order to remove view symbols, and replace them with target relations. To do this, we apply a number of transformations that guarantee that the rewritten mapping yields equivalent results wrt to input one, in the sense discussed in Section 4.

Algorithm 1 reports the pseudocode of our unfolding algorithm *UnfoldDependencies*. To define the algorithm, we use the standard unfolding algorithm for (positive) conjunctive views, *unfoldView* [13], as a building block.

The main intuition behind the algorithm is easily stated: it works with a set of dependencies, called  $\Sigma$ , initialized as  $\Sigma_{SV} \cup \Sigma_V$ , and progressively transforms

---

**Algorithm 1** *UnfoldDependencies*( $\Sigma_{SV}, \Sigma_V, \Upsilon_V$ )

---

```

 $\Sigma := \Sigma_{SV} \cup \Sigma_V$ 
repeat
  for all  $d \in \Sigma$  do
    // Transformation 1.
    if  $d$  contains a positive derived atom  $L$  then
      for all view definition  $v_i$  of  $L$  in  $\Upsilon_V$  do
         $\Sigma := \Sigma \cup \{\text{unfoldView}(L, d, v_i)\}$ 
      end for
       $\Sigma := \Sigma - \{d\}$ 
    end if
    // Transformation 2.
    if  $d$  is a ded containing a negative derived atom  $\neg L(\bar{x}_i, \bar{y}_i)$  in  $\psi_j(\bar{x}, \bar{y}_j)$ 
    then
      let  $TGD_k$  be a new relation symbol
       $d := \phi(\bar{x}) \rightarrow \dots \vee (\psi_j(\bar{x}, \bar{y}_j) - \{\neg L(\bar{x}_i, \bar{y}_i)\}) \cup \{TGD_k(\bar{x}_i, \bar{y}_i)\} \vee \dots$ 
       $d^1 := TGD_k(\bar{x}_i, \bar{y}_i) \wedge L(\bar{x}_i, \bar{y}_i) \rightarrow \perp$ 
       $\Sigma := \Sigma \cup \{d^1\}$ 
    end if
    // Transformation 3.
    if  $d$  is a denial  $\phi(\bar{x}) \rightarrow \perp$  containing a negative atom  $\neg L(\bar{x}_i)$  in  $\phi(\bar{x})$ 
    then
       $d := \phi(\bar{x}) - \{\neg L(\bar{x}_i)\} \rightarrow L(\bar{x}_i)$ 
    end if
    // Transformation 4.
    if  $d$  is a ded containing a negative atom  $\neg L(\bar{x}_i)$  in  $\phi(\bar{x})$  then
       $d := \phi(\bar{x}) - \{\neg L(\bar{x}_i)\} \rightarrow \psi_1(\bar{x}, \bar{y}_1) \vee \dots \vee \psi_n(\bar{x}, \bar{y}_n) \vee L(\bar{x}_i)$ 
    end if
  end for
until fixpoint
 $\Sigma_{ST} :=$  the set of s-t deds in  $\Sigma$ 
 $\Sigma_T :=$  the set of target deds and denials in  $\Sigma$ 

```

---

469 this set until a fixpoint is reached. Note that it always terminates, since we  
 470 assume the view definitions are not recursive. The algorithm employs four main  
 471 transformations in order to remove derived atoms from the dependencies of  $\Sigma$ :

472 **Transformation 1:** First, whenever a positive derived atom  $L(\bar{x}_i)$  is found  
 473 in a dependency  $d$ , the algorithm uses the standard view unfolding algorithm  
 474 as a building block in order to replace  $L(\bar{x}_i)$  by its view definitions. The al-  
 475 ternative definitions that may exist for a single view are handled in parallel.  
 476 Therefore, the unfolding algorithm replaces dependency  $d$  with a set of depen-  
 477 dependencies  $\{d'_1, d'_2, \dots\}$ , where each  $d'_i$  is like  $d$  after replacing  $L(\bar{x}_i)$  by one of its  
 478 definitions. To see an example, consider tgds  $m_0$  and  $m_2$ , and views Average  
 479 and Problematic:

$$\begin{aligned} m_0 : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr > 4, mingr < 9 \rightarrow \text{Average}(id, name), \text{Worker}(id, name) \\ m_2 : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr \leq 4 \rightarrow \text{Problematic}(id, name), \text{Worker}(id, name) \\ v_5 : & \text{Problematic}(id, name) \Leftarrow \text{Worker}(id, name), \text{Penalized}(id, year) \\ v_6 : & \text{Problematic}(id, name) \Leftarrow \text{Worker}(id, name), \neg \text{PositiveEval}(id, year, sinc) \\ v_8 : & \text{Average}(id, name) \Leftarrow \text{Worker}(id, name), \neg \text{Outstanding}(id, name), \\ & \quad \neg \text{Problematic}(id, name) \end{aligned}$$

480  
 481 Standard unfolding with  $v_8$  changes  $m_0$  as follows:

$$\begin{aligned} m_0 : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr > 4, mingr < 9 \rightarrow \text{Worker}(id, name), \neg \text{Outstanding}(id, name), \\ & \quad \neg \text{Problematic}(id, name) \end{aligned}$$

482  
 483 Standard unfolding with  $v_5$  and  $v_6$ , respectively, changes  $m_2$  as follows:

$$\begin{aligned} m_{2a} : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr \leq 4 \rightarrow \exists year' : \text{Worker}(id, name), \text{Penalized}(id, year') \\ m_{2b} : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr \leq 4 \rightarrow \exists year', sinc' : \text{Worker}(id, name), \neg \text{PositiveEval}(id, year', sinc') \end{aligned}$$

484  
 485 Note that a single unfolding step might not be enough to fully remove all  
 486 positive derived atoms, so successive applications of this first transformation  
 487 may be required. In the example above, unfolding  $m_0$ ,  $m_{2a}$  and  $m_{2b}$  with view  
 488 Worker yields:

$$\begin{aligned} m_0 : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr > 4, mingr < 9 \rightarrow \text{Employees}(id, name), \neg \text{Outstanding}(id, name), \\ & \quad \neg \text{Problematic}(id, name) \\ m_{2a} : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr \leq 4 \rightarrow \exists year' : \text{Employees}(id, name), \text{Penalized}(id, year') \\ m_{2b} : & S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ & \quad maxgr \leq 4 \rightarrow \exists year', sinc' : \text{Employees}(id, name), \neg \text{PositiveEval}(id, year', sinc') \end{aligned}$$

489  
 490 **Transformation 2:** The second, and most important transformation, han-  
 491 dles negated view atoms  $\neg L(\bar{x}_i, \bar{y}_i)$  in tgd conclusions, e.g. Outstanding and

492 Problematic in  $m_0$  and PositiveEval in  $m_{2b}$ ; we cannot directly unfold a negated  
 493 derived atom of the conclusion in order to have an equivalent tgdt; we need a  
 494 way to express more appropriately the intended semantics, i.e., the fact that  
 495 the tgdt should be fired only if it is not possible to satisfy  $L(\bar{x}_i, \bar{y}_i)$ ; to express  
 496 this, we replace the negated atom from the conclusion (let us focus on  $m_0$  and  
 497 Outstanding for now) with a new relation symbol  $TGD_i(\bar{x}_i, \bar{y}_i)$

$$m_0^1 : S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ maxgr > 4, mingr < 9 \rightarrow Employees(id, name), TGD_0(id, name), \\ \neg \text{Problematic}(id, name)$$

498  
 499 and introduce a new dependency  $d^1$ , which states that  $d$  should fire only if it is  
 500 not possible to satisfy  $L(\bar{x}_i, \bar{y}_i)$ , by means of a denial constraint:

$$m_0^2 : TGD_0(id, name), \text{Outstanding}(id, name) \rightarrow \perp$$

501  
 502 Note that since a tgdt may have more than one negated atom in the conclu-  
 503 sion, the second transformation may have to be applied multiple times. The full  
 504 result of the transformation when successively applied to  $m_0$ ,  $m_{2a}$  and  $m_{2b}$  is  
 505 the following:

$$m_0^1 : S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ maxgr > 4, mingr < 9 \rightarrow Employees(id, name), TGD_0(id, name) \\ m_0^2 : TGD_0(id, name), \text{Outstanding}(id, name) \rightarrow \perp \\ m_0^3 : TGD_0(id, name), \text{Problematic}(id, name) \rightarrow \perp \\ m_{2a}^1 : S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ maxgr \leq 4 \rightarrow \exists year' : Employees(id, name), Penalized(id, year') \\ m_{2b}^1 : S\text{-WorkerGrades}(id, yr, gr, sinc), S\text{-Stats}(id, name, mingr, maxgr), \\ maxgr \leq 4 \rightarrow \exists year', sinc' : Employees(id, name), TGD_1(id, year', sinc') \\ m_{2b}^2 : TGD_1(id, year, sinc), \text{PositiveEval}(id, year, sinc) \rightarrow \perp$$

506  
 507 **Transformation 3:** The third transformation consists of moving negated  
 508 atoms of the form  $\neg L(\bar{x}_i)$  in the premise of a denial constraint  $d$  to its conclusion,  
 509 in order to remove the negation. To see an example of this, we advance in the  
 510 rewriting of  $m_0^2$ ; transformation 1 needs to be applied again in order to unfold  
 511 the Outstanding atom:

$$m_0^2 : TGD_0(id, name), \text{Worker}(id, name), \neg \text{NegativeEval}(id, year), \\ \neg \text{Warned}(id, date) \rightarrow \perp$$

512  
 513 However, the negative atoms may be moved easily to the conclusion, to yield  
 514 a target dtgd:

$$m_0^2 : TGD_0(id, name), \text{Worker}(id, name) \rightarrow \exists year, date : \text{NegativeEval}(id, year) \\ \vee \text{Warned}(id, date)$$

515  
 516 To complete the rewriting of  $m_0^2$ , the unfolding algorithm would keep on  
 517 applying transformations 1 and 2.

**Transformation 4:** The fourth and final transformation is a variation of transformation 3 that is applied to ded. The only difference is that the atoms being moved from the premise are disjuncted to the current contents of the conclusion instead of replacing it.

The complete rewriting of the running example is reported in Appendix B.

### 7.1. Correctness Result

We are now ready to state our main result about the correctness of the rewriting algorithm. Before we do that, we should make more precise the schemas that are involved in the translation. We start with a target schema,  $\mathbf{T}$ , but during the rewriting we enrich it with new relation symbols,  $TGD_0, TGD_1, \dots$ , in order to be able to correctly specify denials. We call the resulting schema  $\mathbf{T}'$ .

**Theorem 2 (Correctness).** *Given a source-to-ontology scenario  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$  with non-recursive view definitions  $\Upsilon_V$ , then:*

- (a) *algorithm `UnfoldDependencies` always terminates;*
- (b) *when it does not fail, it computes a correct source-to-target rewritten scenario with ded  $\mathcal{M}_{ST'} = \{\mathbf{S}, \mathbf{T}', \Sigma_{ST'}, \Sigma_{T'}\}$ , where  $\mathbf{T}'$  is obtained from  $\mathbf{T}$  by enriching it with a finite set of new relation symbols  $TGD_0, TGD_1, \dots$ .*

## 8. A Restricted Case

Theorem 2 shows that Algorithm 1 is correct. However, we also know that it may incur significant scalability issues, that we discuss in Section 9. This leaves us with a crucial question: is it possible to find a view-definition language that is at the same time more expressive than plain conjunctive queries, and computes correct rewritings in terms of embedded dependencies, i.e. tgds, egds, and standard denial constraints only?

In this section, we show that such a view-definition language exists, and corresponds to non-recursive Datalog with a limited negation. To be more precise, we limit negation in such a way that: (i) we disallow some pathological patterns within view definitions with negations; (ii) keys and functional dependencies — i.e. egds — are defined only for views whose definition does not depend on negated atoms.

**Definition 4 (Negation-Safe View Language).** Given a set of non-recursive view definitions,  $\Upsilon_V$ , we say that these are *negation-safe* if the following occur:

1. there is no view  $V_i$  that negatively depends on a view  $V_j$  that in turn negatively depends on two negated atoms;
2. keys and functional dependencies are defined only for views whose definitions do not contain negated atoms.

In essence, item 1 above disallows very specific view-definition patterns, like the one below:

$$\begin{aligned} v_1 : V_1(x, y) &\Leftarrow T_1(x, y), \neg V_2(x, y) \\ v_2 : V_2(x, y) &\Leftarrow T_2(x, y), \neg V_3(x, y), \neg V_4(x, y) \\ &\dots \end{aligned}$$

Item 1 prohibits the definition of keys on views  $V_1, V_2$  that contain negated views in their definitions. We can show that the condition in Definition 4 is a sufficient condition that guarantees that Algorithm 1 returns a set of embedded dependencies, and does not generate ded. s.

**Theorem 3 (Restriction).** *Given a source-to-ontology scenario  $\mathcal{M}_{SV} = \{\mathcal{S}, \mathcal{V}, \Sigma_{SV}, \Sigma_V\}$  with view definition  $\Upsilon_V$ , assume  $\Upsilon_V$  conforms to the restrictions in Definition 4. Call  $\mathcal{M}_{ST'}^{emb} = \{\mathcal{S}, \mathcal{T}', \Sigma_{ST'}, \Sigma_{T'}\}$ , the source-to-target rewritten scenario computed by algorithm *UnfoldDependencies*, where  $\mathcal{T}'$  is obtained from  $\mathcal{T}$  by enriching it with a finite set of new relation symbols  $TGD_0, TGD_1, \dots$ . Then  $\mathcal{M}_{ST'}^{emb}$  only contains embedded dependencies (i.e. tgds, egds, and denial constraints).*

Theorem 3 guarantees that, under the conditions of Definition 4, the rewritten source-to-target mapping is a set of standard tgds, egds, and denial constraints. This has important implications on the scalability of the data-exchange process, as we discuss in the next section.

## 9. The Chase Engine

Once we have computed our source-to-target mapping, we can concretely attempt the actual data exchange, and move data from the source database to the target. The standard way to do this corresponds to running the well known *chase* [3] procedure, i.e. an operational semantics for embedded dependencies that we discuss in the following.

### 9.1. The Chase

Given a vector of variables  $\bar{v}$ , an *assignment* for  $\bar{v}$  is a mapping  $a : \bar{v} \rightarrow \text{CONST} \cup \text{NULLS}$  that associates with each universal variable a constant in  $\text{CONST}$ , and with each existential variable either a constant or a labeled null. Given a formula  $\phi(\bar{x})$  with free variables  $\bar{x}$ , and an instance  $I$ , we say that  $I$  *satisfies*  $\phi(a(\bar{x}))$  if  $I \models \phi(a(\bar{x}))$ , according to the standard notion of logical entailment.

Of the many variants of the chase, we consider the *naive chase* [4]. We first introduce the notions of *chase steps* for tgds, egds, and denial constraints, and then the notions of a chase sequence and of a chase result.

**Chase Step for Tgds:** Given instances  $I, J$ , a tgd  $\phi(\bar{x}) \rightarrow \exists \bar{y}(\psi(\bar{x}, \bar{y}))$  is fired for all assignments  $a$  such that  $I \models \phi(a(\bar{x}))$ ; to fire the tgd,  $a$  is extended to  $\bar{y}$  by injectively assigning to each  $y_i \in \bar{y}$  a fresh null, and then adding the facts in  $\psi(a(\bar{x}), a(\bar{y}))$  to  $J$ . To give an example, consider the following tgd:

$$m. \text{Driver}(\text{name}, \text{plate}) \rightarrow \exists Bdate, \text{CarId}: \text{Person}(\text{name}, Bdate, \text{CarId}), \\ \text{Car}(\text{CarId}, \text{plate})$$

During the chase, the source tuple  $Driver(Jim, abc123)$  will generate the two target tuples  $Person(Jim, N_1, C1)$ , and  $Car(C1, abc123)$ , where  $N_1, C1$  are fresh labeled nulls.

*Chase Step for Egds:* To chase an egd  $\forall \bar{x} : \phi(\bar{x}) \rightarrow x_i = x_j$  over an instance  $J$ , for each assignment  $a$  such that  $J \models \phi(a(\bar{x}))$ , if  $a(x_i) \neq a(x_j)$ , the chase tries to equate the two values. We distinguish two cases: (i) both  $a(x_i)$   $a(x_j)$  are constants; in this case, the chase procedure *fails*, since it attempts to identify two different constants; (ii) at least one of  $a(x_i)$ ,  $a(x_j)$  is a null, say  $a(x_i)$ ; in this case chasing the egd generates a new instance  $J'$  obtained from  $J$  by replacing all occurrences of  $a(x_i)$  by  $a(x_j)$ . To give an example, consider egd  $e_1$ :

$$e_1. Person(name, b, c), Person(name, b', c') \rightarrow (b = b') \wedge (c = c')$$

Assume two tuples have been generated by chasing the tgds,  $Person(Jim, 1980, N_4)$ ,  $Person(Jim, N_5, N_6)$ , chasing the egd has two different effects: (i) it replaces nulls by constants; in our example, it equates  $N_5$  to the constant 1980, based on the same value for the key attribute,  $Jim$ ; (ii) on the other side, the chase might equate nulls; in our example, it equates  $N_4$  to  $N_6$ , to generate a single tuple  $Person(Jim, 1980, N_4)$ .

*Chase Step for Denial Constraints:* Denial constraints can only generate failures. More specifically, the chase of a denial constraint  $\forall \bar{x} : \phi(\bar{x}) \rightarrow \perp$  over an instance  $J$  fails whenever there exists an assignment  $a$  such that  $J \models \phi(a(\bar{x}))$

Given a mapping scenario  $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{ST}, \Sigma_T)$  and instance  $I$ , a *chase sequence* is a sequence of instances  $J_0 = I, J_1, \dots, J_k \dots$ , such that each  $J_i$  is generated by a chase step with  $\Sigma_{ST} \cup \Sigma_T$  over  $J_{i-1}$ . The *chase* of  $\Sigma_{ST} \cup \Sigma_T$  is an instance  $J_m$  such that no chase step is applicable. Notice that the chase may not terminate [3]. This may happen, for example, in the case of recursive target tgds. However, if it terminates, then  $J_m$  is a solution for  $\mathcal{M}$  and  $I$ , called a *canonical solution*.

Any canonical solution is a universal solution [3]. Since all solutions obtained by using the naive chase are equal up to the renaming of nulls, we often speak of *the canonical universal solution*.

## 9.2. A Greedy Chase

For the purpose of this work, we adopt the chase engine developed within the LLUNATIC project [10, 18], that is freely available.<sup>3</sup> The chase engine was developed to guarantee high scalability, even for large sets of embedded dependencies, and large source instances.

Therefore, we expect that the data-exchange step can be completed quite efficiently under the conditions of Definition 4 and Theorem 3, i.e. when the rewriting algorithm returns a set of standard embedded dependencies.

Things change quite dramatically when the rewriting algorithm returns a set of ded. As we noticed in Section 5, ded. have a perverse effect on the

<sup>3</sup><http://db.unibas.it/projects/llunatic>



complexity of computing solutions. Given an instance  $I$ , a set of deds may have a number of solutions over  $I$  that is exponential in the size of  $I$ .

Intuitively, the chase also changes. In fact, the chase of deds generates *chase trees*, not chase sequences. Consider the following example, where we are given two deds:

$$\begin{aligned} m_{d_1} : \forall x : S_1(x) \rightarrow (\exists y : T_1(x, y)) \vee (T_2(x, x)) \\ m_{d_2} : \forall x : S_2(x) \rightarrow (\exists y : T_3(x, y), T_3(y, x)) \vee (\exists z : T_4(x, z)) \end{aligned}$$

We start chasing these on source instance  $I = \{S_1(a), S_2(b)\}$ . A first assignment  $a(x) = 'a'$  such that  $I \models S_1(a(x))$  is found, and therefore we may fire  $m_{d_1}$ . However, two alternative target instances may be generated, namely  $J_1 = \{T_1(a, N_1)\}$  and  $J_2 = \{T_2(a, a)\}$ . These need to be considered in parallel, and therefore a chase tree rooted at  $J_0 = \emptyset$ , i.e. the empty target instance, with children  $J_1, J_2$  is built. To proceed with the chase, we need to inspect every leaf, and apply successive chase steps. This happens with assignment  $a(x) = 'b'$ , according to which the premise of the second ded is satisfied by  $I$ . It is easy to see that we have two different ways to satisfy the ded, and therefore we end up with a chase tree with four leaves, each of which is a solution for this simple scenario. These, together, form a universal model set for the deds, as follows:

$$\mathbf{J} = \left\{ \begin{array}{ll} \{T_1(a, N_1), T_3(b, N_2), T_3(N_2, b)\}, & \{T_2(a, a), T_3(b, N_2), T_3(N_2, b)\}, \\ \{T_1(a, N_1), T_4(b, N_4)\}, & \{T_2(a, a), T_4(b, N_5)\} \end{array} \right\}$$

Recall that there are cases in which the size of the chase tree is exponential in the size of the input instance  $I$ . As a consequence, there is little hope that we are able to perform this parallel chase in a scalable way.

Recall, however, that our rewriting algorithm follows a best-effort approach. Along the same lines, we may consider giving up the idea of generating the entire tree, and rather concentrate on some of its branches, following a greedy strategy. To be more precise, we notice that the four leaves of the chase tree correspond each to the canonical solution of one of the following four sets of (standard) tgds:

$$\begin{aligned} \Sigma_{11} : \quad & m_{11} : \forall x : S_1(x) \rightarrow (\exists y : T_1(x, y)) \\ & m_{21} : \forall x : S_2(x) \rightarrow (\exists y : T_3(x, y), T_3(y, x)) \\ \Sigma_{12} : \quad & m_{11} : \forall x : S_1(x) \rightarrow (\exists y : T_1(x, y)) \\ & m_{22} : \forall x : S_2(x) \rightarrow (\exists z : T_4(x, z)) \\ \Sigma_{21} : \quad & m_{12} : \forall x : S_1(x) \rightarrow T_2(x, x) \\ & m_{21} : \forall x : S_2(x) \rightarrow (\exists y : T_3(x, y), T_3(y, x)) \\ \Sigma_{22} : \quad & m_{12} : \forall x : S_1(x) \rightarrow T_2(x, x) \\ & m_{22} : \forall x : S_2(x) \rightarrow (\exists z : T_4(x, z)) \end{aligned}$$

For example,  $\Sigma_{11}$  generates those solutions that were generated by the chase of  $m_{d_1}, m_{d_2}$  along those branches of the chase tree in which the first conjunct of both deds was always chosen. Similarly for the others.



We call these the *greedy scenarios* associated with a mapping scenario with ded. Greedy scenarios do not generate all of the canonical solutions associated with a mapping scenario with ded. In fact, they are not able to capture the chase strategies in which the same ded is fired according to the first conjunct at some step, and according to another conjunct at a following step. However, their canonical solutions can be computed in a scalable way.

This justifies our chase strategy with ded:

- (i) given a mapping scenario  $\mathcal{M}^{ded}$  with a set of ded  $\Sigma^{ded}$ , we generate the associated greedy scenarios,  $\mathcal{M}_0^{emb}, \mathcal{M}_1^{emb}, \dots, \mathcal{M}_n^{emb}$ ; each is obtained by picking a different combination of the conjuncts that are present in ded conclusions;
- (ii) given an instance  $I$ , we start chasing the greedy scenarios, one by one, on  $I$ ; as soon as we get a canonical solution  $J_i$  for greedy scenario  $\mathcal{M}_i^{emb}$  and  $I$ , we return  $J_i$  and stop;
- (iii) if every greedy scenario fails on  $I$ , we fail and return no solution.

In the following section, we study the scalability of this approach.

## 10. Experiments

We implemented a prototype of our rewriting algorithm in Java. In order to execute the mappings, we used the free and highly scalable chase engine LLUNATIC [18]. We performed our experiments on an Intel core i7 machine with a 2.6 GHz processor, 8 GB of RAM, and running MacOSX. We used PostgreSQL 9.2.1 (x64 version) as the DBMS.

**Scenarios** We used three different datasets from which we derived a number of different scenarios:

- (a) WORKERS is obtained by applying the unfolding algorithm to the source-to-ontology mapping scenario described in the Appendix B. This is a ded-based scenario with 3 source and 15 target tables. It contains 23 ded that generate 20 different greedy scenarios.
- (b) Recall that scenarios with ded are chased by successively chasing their greedy versions. Since we are also interested in studying how each of these greedy scenarios (without ded) impacts performance, in our tests we also consider the first greedy scenario generated for WORKERS, and denote it by WORKERS-GREEDY-1. This has 10 st-tgds, 4 target tgds, 3 target egds, and 7 denial constraints.
- (c) EMPLOYEES is a traditional schema mapping scenario based on the example proposed in [11]. It contains 2 source and 10 target tables, 9 st-tgds, 5 target tgds, 2 target egds, and 2 denial constraints.
- (d) To study the impact of egds on the rewriting algorithm and on the chase, we also consider an egd-free version of EMPLOYEES, called EMPLOYEES No-EGD.
- (e) Finally, we want to test the scalability of the rewriting algorithm. For this purpose, we take a fully synthetic dataset, called SYNTHETIC. Based on this,

we generated seven different scenarios, with a number of dependencies ranging from 50 to 30K dependencies.

**Effectiveness** To measure the effectiveness of our approach, we compared the size of the source-to-ontology mapping that users need to specify for the various scenarios, to the size of the actual source-to-target scenario generated by our rewriting. As a measure of the size of a scenario, we took the number of nodes and edges of the *dependency graph* [3], i.e. the graph in which each atom of a dependency is a node, and there is an edge from node  $n_1$  to node  $n_2$  whenever the corresponding atoms share a variable. Intuitively, the higher the complexity of this graph, the more complicated it is to express the mapping. Figure 6a reports the results for 5 scenarios. In all scenarios there was a considerable increase in the size of the dependency graph (up to 70%). This is a clear indication that in many cases our approach is more effective with respect to manually developing the source-to-target mapping.

**Scalability of the Rewriting Algorithm** The second set of experiments tests the scalability of our unfolding algorithm on mapping scenarios of a large size. Figure 6b summarizes results of these experiments on scenarios of increasing size. All source-to-ontology tgds in these scenarios have two source relations in the premise and two views in the conclusion. Each view definition has two positive target relational symbols and (if the view has negation) two negated view symbols. For each mapping scenario, 20% of the tgds have no negated atoms, the next 20% have 1 level of negation (i.e. negated atoms that do not depend in turn on other negations), the next 20% have 2 levels of negation, and so on, up to 4 levels of negations. The number of source relations in the mapping scenarios ranges from 10k to 60k, the number of view definitions ranges from 238k to 1428k, and the number of target relations ranges from 228k to 1368k. The reported times are the running times of the unfolding algorithm running in main memory, and do not include disk read and write times. The rewriting algorithm scales nicely to large scenarios.

**Scalability of the Chase** Our final goal is to study the scalability of the chase engine, i.e. how expensive it is to execute the source-to-target rewritten mapping. To do this, we first study the performance of the chase engine on schema mapping scenarios with no deds. This is important, since previous research [5, 19, 6] have shown that some of the existing chase engines hardly scale to large datasets. Figure 6c and 6d report the time needed to compute a solution for four of our scenarios. As expected, scenarios with no egds required lower computing times. However, in the case of egds the chase engine also scaled nicely to databases of 1 million tuples.

To test scenarios with deds, we developed the greedy-chase algorithm described in Section 9.2 on top of LLUNATIC. Recall that, given a mapping scenario with deds, we generate a set of greedy scenarios with embedded dependencies only. The first experiment in this context was to test how many of the 20 greedy scenarios associated to the WORKERS scenario do return a solution.

We first generated four different random source instances and in Figure 6e we report the results. The greedy algorithm generated a solution in all of the four

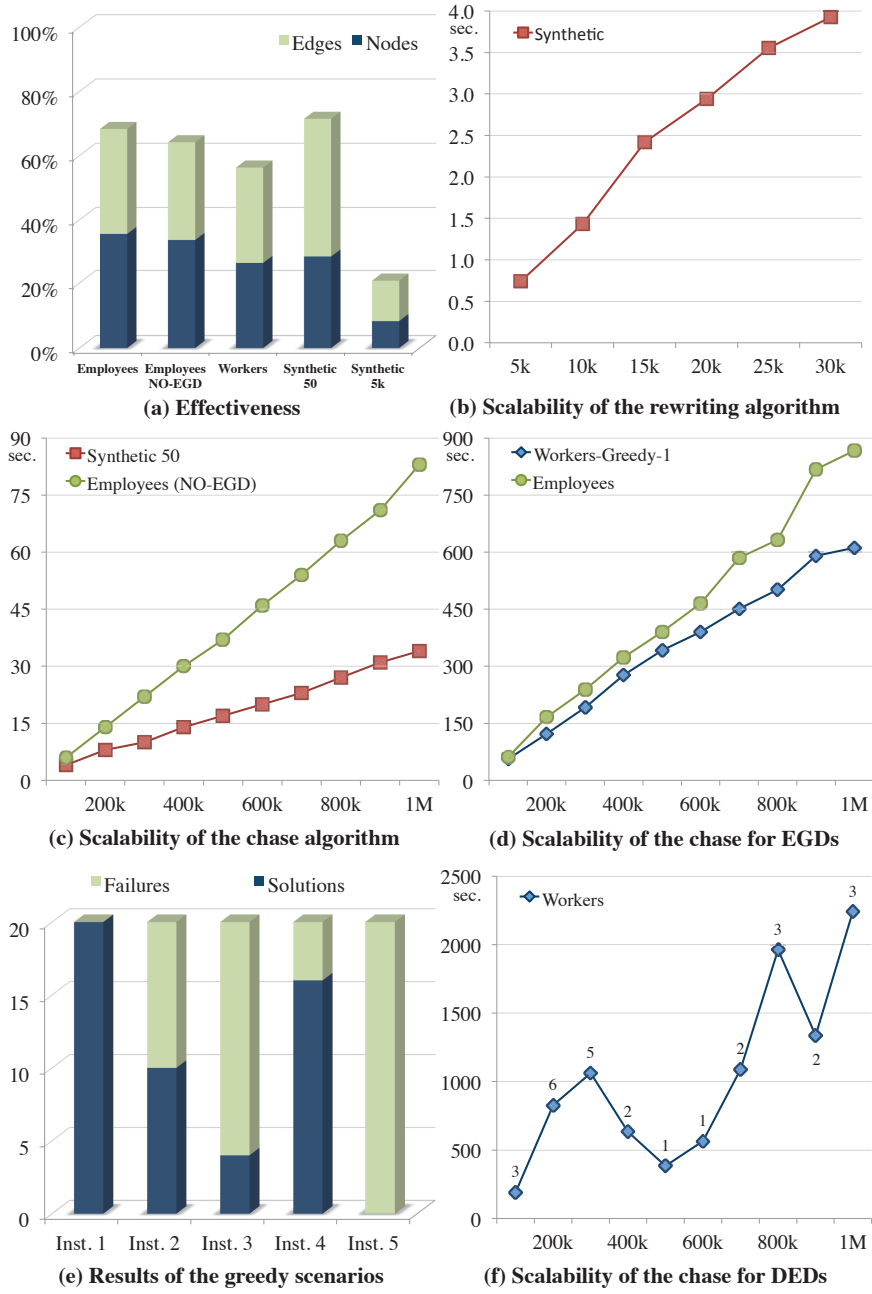


Figure 6: Results of Experiments.

cases. Then, we studied the possible failure conditions, and manually crafted an instance with a high probability of triggering the denial constraints. With this fifth source instance, all of the 20 greedy scenarios failed. Notice that a solution still exists. However, this is not captured by the combinations of atoms in greedy scenarios, and would require the generation of the entire chase tree to be found.

Finally in Figure 6f we report scalability results for the greedy chase algorithm. For each execution we also report the number of greedy scenarios that the chase engine needed to run in order to reach a solution. As can be seen, the chase scales nicely, even with databases of 1 million of tuples. Spikes in computing times are due to the need to execute fewer scenarios before a solution is found. To the best of our knowledge, this is the first scalability result for the chase of disjunctive embedded dependencies.

## 11. Related Work

The standard view unfolding algorithm [13] has been used extensively in data integration as a tool for query answering. In such a setting, users pose queries over a set of heterogeneous sources through a single global schema, which provides a uniform view of all the sources. Mappings between the sources and the global schema are used to rewrite the users' queries in terms of the sources. One way to define these mappings is the so-called global-as-view approach (GAV), in which the global schema is defined as a view over the sources. With this kind of mapping, answering a query posed on the global schema usually reduces to unfolding the view definitions [7] (unless integrity constraints are present in the global schema, which makes answering harder [14]).

Another similar problem is that of accessing data through ontologies, in which users pose queries on an ontology that is defined on top of a set of databases; the ontology plays the role of global schema, and the databases play the role of data sources [20, 21]. The problem we address in this paper, however, is not about using view unfolding to answer queries, but to copy data into a target. As we have discussed in Section 7, standard view unfolding suffices only when the views that define the target conceptual schema in terms of the underlying database are plain conjunctive queries. In the presence of negation, copying data into the target gets more complicated, as negated atoms in mapping conclusions introduce new integrity constraints that standard view unfolding does not handle (intuitively, negated atoms must be kept false during all the process of copying data into the target).

A problem that relates to our use of view unfolding in mappings is that of mapping composition [22, 23]. Composing a mapping between schemas  $A$  and  $B$  with a mapping between schemas  $B$  and  $C$  produces a new mapping between  $A$  and  $C$ . In a sense, our application of view unfolding to the conclusion of a mapping can be seen as a kind of mapping composition; one in which the mapping between the source and the conceptual schema is composed with a second mapping that relates the conceptual schema with the underlying database (i.e. the views). However, mapping composition techniques take into account the

direction of the mapping, that is, one can compose a mapping from  $A$  to  $B$  only with another mapping that goes from  $B$  to some  $C$  in order to get a mapping that goes from  $A$  to  $C$ . In our case, we have a mapping from the source to the conceptual schema and another one from the database to the conceptual schema, which cannot be directly composed.

The introduction of conceptual schemas into the mapping process has also been investigated in [24] with respect to a different problem, i.e. that of generating mappings between databases. Since we assume that source-to-ontology mappings are given as inputs, the techniques developed in [24] can be used as a preliminary step to simplify the mapping specification phase.

Another context where mappings involving conceptual schemas have been studied is that of Semantic Web ontologies; in particular, [25] proposes a technique that translates a set of correspondences between source and target ontologies into a set of SPARQL queries that can then be run against the data source to produce the target's data. Comparing with our approach, we assume that the given mapping is not just a set of correspondences, but a complete declarative mapping expressed as tgds, and we also take into account that the target's conceptual schema is a view of the underlying database.

Mappings between conceptual schemas have also been studied in [26], where the authors propose an approach for finding "semantically similar" associations between two conceptual schemas. These similar associations are then used to generate a mapping. This approach is complementary to ours in the sense that it could be used to generate a semantic-based mapping, which would then be rewritten using the algorithm we present in this paper.

## 12. Conclusion

This paper studies the problem of mapping data in the presence of ontology-based descriptions of the source and target data sources. It shows that employing an expressive view-definition language for the purpose of defining ontologies makes the rewriting process much more complicated than in the case of positive conjunctive views. The paper develops an algorithm to automatically perform the rewriting when views are defined by means of non-recursive Datalog rules with negation. This, in turn, required the adoption of a very expressive mapping language involving disjunctive embedded dependencies.

To handle the increased complexity of this mapping language, we investigated restrictions to the view-definition language that may be handled using standard embedded dependencies (i.e. tgds and egds) for which efficient execution strategies exist. We conducted experiments on large databases and mapping scenarios to show the trade-off between expressibility of the view language and the efficiency of the data exchange step.

As future work, we plan to investigate the use of other execution strategies to perform the actual data-exchange to move data from the source to the target database rather than the greedy chase considered here. We would also like to analyze the applicability of our techniques to ontology based updating, seen as a parallel notion to the classical problem of ontology based querying.

## References

- [1] R. J. Miller, L. M. Haas, M. A. Hernandez, Schema Mapping as Query Discovery, in: VLDB, 2000, pp. 77–99.
- [2] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, R. Fagin, Translating Web Data, in: VLDB, 2002, pp. 598–609.
- [3] R. Fagin, P. Kolaitis, R. Miller, L. Popa, Data Exchange: Semantics and Query Answering, TCS 336 (1) (2005) 89–124.
- [4] B. ten Cate, L. Chiticariu, P. Kolaitis, W. C. Tan, Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries, PVLDB 2 (1) (2009) 1006–1017.
- [5] G. Mecca, P. Papotti, S. Raunich, Core Schema Mappings: Scalable Core Computations in Data Exchange, Inf. Syst 37 (7) (2012) 677–711.
- [6] B. Marnette, G. Mecca, P. Papotti, S. Raunich, D. Santoro, ++SPICY: an opensource tool for second-generation schema mapping and data exchange, PVLDB 4 (11) (2011) 1438–1441.
- [7] M. Lenzerini, Data integration: a Theoretical Perspective, in: PODS, 2002.
- [8] C. Beeri, M. Vardi, A Proof Procedure for Data Dependencies, J.ACM 31 (4) (1984) 718–741.
- [9] F. Geerts, G. Mecca, P. Papotti, D. Santoro, The LLUNATIC Data-Cleaning Framework, PVLDB 6 (9) (2013) 625–636.
- [10] F. Geerts, G. Mecca, P. Papotti, D. Santoro, Mapping and Cleaning, in: ICDE, 2014, pp. 232–243.
- [11] G. Mecca, G. Rull, D. Santoro, E. Teniente, Semantic-Based Mappings, in: ER, 2013, pp. 255–269.
- [12] S. Ceri, G. Gottlob, L. Tanca, What you Always Wanted to Know About Datalog (And Never Dared to Ask), IEEE TKDE 1 (1) (1989) 146–166.
- [13] L. Sterling, E. Y. Shapiro, The Art of Prolog: Advanced Programming Techniques, MIT Press, 1994.
- [14] A. Cali, D. Calvanese, G. De Giacomo, M. Lenzerini, Data integration under integrity constraints, Inf. Syst 29 (2) (2004) 147–163.
- [15] A. Deutsch, A. Nash, J. Remmel, The chase revisited, in: PODS ’08, 2008, pp. 149–158.
- [16] R. Fagin, P. Kolaitis, A. Nash, L. Popa, Towards a Theory of Schema-Mapping Optimization, in: PODS, 2008, pp. 33–42.

- 844 [17] A. Queral, E. Teniente, Reasoning on uml class diagrams with ocl con-  
845 straints, in: ER, 2006, pp. 497–512.
- 846 [18] F. Geerts, G. Mecca, P. Papotti, D. Santoro, That’s All Folks! LLUNATIC  
847 Goes Open Source, PVLDB 7 (13) (2014) 1565–1568.
- 848 [19] G. Mecca, P. Papotti, S. Raunich, M. Buoncristiano, Concise and Expres-  
849 sive Mappings with +SPICY, PVLDB 2 (2) (2009) 1582–1585.
- 850 [20] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati,  
851 Linking data to ontologies, J. Data Semantics 10 (2008) 133–173.
- 852 [21] D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati, View-based query  
853 answering in description logics: Semantics and complexity, J. Comput.  
854 Syst. Sci. 78 (1) (2012) 26–46.
- 855 [22] R. Fagin, P. G. Kolaitis, L. Popa, W. C. Tan, Composing schema mappings:  
856 Second-order dependencies to the rescue, ACM TODS 30 (4) (2005) 994–  
857 1055.
- 858 [23] A. Nash, P. A. Bernstein, S. Melnik, Composition of mappings given by  
859 embedded dependencies, ACM Trans. Database Syst. 32 (1) (2007) 4.
- 860 [24] Y. An, A. Borgida, R. Miller, J. Mylopoulos, A Semantic Approach to  
861 Discovering Schema Mapping Expressions, in: ICDE, 2007, pp. 206–215.
- 862 [25] C. R. Rivero, I. Hernández, D. Ruiz, R. Corchuelo, Generating sparql exe-  
863 cutable mappings to integrate ontologies, in: ER, 2011, pp. 118–131.
- 864 [26] Y. An, I.-Y. Song, Discovering semantically similar associations (sesa) for  
865 complex mappings between conceptual models, in: ER, 2008, pp. 369–382.

## Appendix A. Proofs of the Theorems

**Theorem 1** *There exist a source-to-ontology mapping scenario  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$  with view definition  $\Upsilon_V$ , and an instance  $I$ , such that  $\mathcal{M}_{SV}$  and  $I$  admit a universal solution  $J_V \in \text{USol}(\mathcal{M}_{SV}, I)$ , and there exists no source-to-target scenario  $\mathcal{M}_{ST}$  composed of embedded dependencies (tgds and egds) such that  $\mathcal{M}_{ST}$  and  $I$  admit a solution  $J_T$ , and  $J_V = \Upsilon(J_T)$ .*

*Proof:* Consider the following scenario. The source database contains a single table,  $S(A)$ , the target database a single table,  $T(A)$ , and we have two views,  $V_1(A), V_2(A)$ , defined as follows:

$$\Upsilon_V = \left\{ \begin{array}{l} V_1(x) \Leftarrow T(x) \\ V_2(x) \Leftarrow T(x), \neg V_1(x) \end{array} \right\}$$

The source-to-ontology mappings are the following ( $\Sigma_V$  is empty):

$$\Sigma_{SV} = \left\{ \begin{array}{l} S(x) \rightarrow V_1(x) \\ S(x) \rightarrow V_2(x) \end{array} \right\}$$

On instance  $I = \{S(a)\}$ ,  $\Sigma_{SV}$  has a universal solution  $J_V = \{V_1(a), V_2(a)\}$ .

We now prove that there exists no target instance  $J_T$  such that  $\Upsilon_V(J_T) = J_V$ . The view definitions in  $\Upsilon_V$  are such that, for any target instance  $J$ ,  $\Upsilon_V(J)$  will not contain tuples  $V_1(c), V_2(c)$  for some constant  $c$ .

Since  $J_T$  does not exist, there is no source-to-target rewriting  $\mathcal{M}_{ST}$  that may generate it as a universal solution for  $I$ , and the claim is proven.  $\square$

**Theorem 2** *Given a source-to-ontology scenario  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$  with non-recursive view definitions  $\Upsilon_V$ , then:*

- (a) *algorithm `UnfoldDependencies` always terminates;*
- (b) *when it does not fail, it computes a correct source-to-target rewritten scenario with deds  $\mathcal{M}_{ST'} = \{\mathbf{S}, \mathbf{T}', \Sigma_{ST'}, \Sigma_{T'}\}$ , where  $\mathbf{T}'$  is obtained from  $\mathbf{T}$  by enriching it with a finite set of new relation symbols  $TGD_0, TGD_1, \dots$ .*

*Proof:* Let us first prove termination, and then correctness.

*Termination* — The proof of part a. depends on the fact that the view definitions in  $\Upsilon_V$  are non-recursive by hypothesis. As a consequence, the set of view symbols,  $V_1, V_2, \dots, V_k$  can be stratified, i.e. it can be partitioned in a sequence of subsets called *strata* such that any view that belongs to stratum  $i$  only depends directly or indirectly on those that appear in strata  $1, 2, \dots, i-1$ .

Algorithm *UnfoldDependencies* is composed of a main loop, and 4 different transformations (*Transformation 1.* to *4.*) that are applied to all dependencies in the current set. The loop stops when a fixpoint is reached. The effects of the various transformations are as follows:



- 895 • *Transformation 1.* unfolds view definition within a dependency  $d$ , i.e. it  
896 replaces a positively occurring view symbol by its definition; therefore, it  
897 removes a view symbol in stratum  $i$  and replaces it with target symbols  
898 or views that belong to strata up to  $i - 1$ ;
- 899 • *Transformation 2.* removes negatively derived atoms from dependency  
900 conclusions, and adds new dependencies;
- 901 • *Transformations 3. and 4.* move negated atoms from a dependency  
902 premise to its conclusion.

903 Given a set of dependencies,  $\Sigma$ , we assign an integer score to it, based on the  
904 following function: the score for  $\Sigma$  is the sum of the scores for its dependencies.  
905 For each dependency, it is the sum of the scores of its atoms that contain view  
906 symbols. With a positive atom  $V(\bar{x}, \bar{y})$  it is associated an integer score  $k^i$ , where  
907  $i$  is the stratum of  $V$ . With a negative atom  $\neg V(\bar{x}, \bar{y})$  it is associated an integer  
908 score  $k^i + 1$ , where  $i$  is again the stratum of  $V$ . It remains to define the value  
909 of  $k$ . Call  $n$  the maximum number of view symbols that appear in the body of  
910 a view definition of  $\Upsilon_V$ . Then  $k = n + 1$ .

911 It is easy to see that the four transformations monotonically decrease the  
912 score of  $\Sigma$ . In fact:

- 913 • *Transformation 1.* replaces positive view atoms from stratum  $i$  by less  
914 than  $k$  view atoms that belong at most to stratum  $i - 1$ ;
- 915 • *Transformation 2.* removes a negated atom of stratum  $i$  from  $d$ , and  
916 introduces a new dependency  $d^1$  that (only) contains a positive atom of  
917 the same stratum;
- 918 • *Transformations 3. and 4.* replace a negated atom of stratum  $i$  within  $d$   
919 by a positive atom of the same stratum in  $d^1$ .

920 Since each iteration of the cycle monotonically reduces the score of  $\Sigma$ , and this  
921 is initially finite, then the number of iterations is bounded, and the algorithm  
922 terminates.

923 *Correctness* — To prove part b., i.e. that the rewritten scenario is correct, we  
924 need to show that the rewriting algorithm is sound wrt the view definitions. This  
925 guarantees that whenever we obtain a solution to the rewritten source-to-target  
926 mapping, we can apply the view definitions to obtain an instance of the ontology  
927 that is a solution to the source-to-ontology mapping. To prove soundness, we  
928 need to prove that the four transformations are sound with respect to the view  
929 definitions.

930 We first notice that *Transformation 1.* corresponds to the standard view  
931 unfolding procedure, which is known to be sound.

*Transformation 3. and 4.* generate dependencies that are logically equivalent  
to the original ones. In *Transformation 3.*, we turn  $\phi(\bar{x}) \wedge \neg L(\bar{x}) \rightarrow \perp$  into  
 $\phi(\bar{x}) \rightarrow L(\bar{x})$ . Call  $a$  the formula  $\phi(\bar{x})$ ,  $b$  atom  $L(\bar{x})$ , then we have that:

$$a \wedge \neg b \rightarrow \perp \equiv \neg(a \wedge \neg b) \equiv \neg a \vee b \equiv a \rightarrow b$$

Similarly, in *Transformation 4.*, we turn  $\phi(\bar{x}) \wedge \neg L(\bar{x}) \rightarrow \bigvee L_i(\bar{x}, \bar{y})$  into  $\phi(\bar{x}) \rightarrow \bigvee L_i(\bar{x}, \bar{y}) \vee L(\bar{x})$ . Call  $a$  the formula  $\phi(\bar{x})$ ,  $b$  atom  $L(\bar{x})$ , and  $c$  the formula  $\bigvee L_i(\bar{x}, \bar{y})$ . Then we have that:

$$a \wedge \neg b \rightarrow c \equiv \neg(a \wedge \neg b) \vee c \equiv \neg a \vee b \vee c \equiv \neg a \vee (b \vee c) \equiv a \rightarrow b \vee c$$

We only need to discuss *Transformation 2.*. This takes a ded of this form:

$$d : \forall x : \phi(\bar{x}) \rightarrow \exists \bar{y} : \bigvee \psi_i(\bar{x}, \bar{y}) \vee (R_0(\bar{x}, \bar{y}) \wedge \dots \wedge \neg L(\bar{x}, \bar{y}) \wedge \dots R_k(\bar{x}, \bar{y}))$$

with a negated  $\neg L(\bar{x}, \bar{y})$  atom in one of its conjuncts, and replaces it by two dependencies. The first one is obtained from  $d$  by replacing  $\neg L(\bar{x}, \bar{y})$  by a new atom  $TGD_i(\bar{x}, \bar{y})$ , where  $TGD_k$  is a new relation symbol:

$$d' : \forall x : \phi(\bar{x}) \rightarrow \exists \bar{y} : \bigvee \psi_i(\bar{x}, \bar{y}) \vee (R_0(\bar{x}, \bar{y}) \wedge \dots \wedge TGD_i(\bar{x}, \bar{y}) \wedge \dots R_k(\bar{x}, \bar{y}))$$

The second one has the form:

$$d^1 : \forall x, y : L(\bar{x}, \bar{y}), TGD_i(\bar{x}, \bar{y}) \rightarrow \perp$$

It is easy to see that any solution for  $d', d^1$  is also a solution for  $d$ . In fact, any solution for  $d', d^1$  must be such that, for any homomorphisms  $h$ , facts  $h(TGD_i(\bar{x}, \bar{y})), h(L(\bar{x}, \bar{y}))$  are not present at the same time. This implies that either the premise of  $d$  is true according to  $h$ , and  $h(L(\bar{x}, \bar{y}))$  is false, or the opposite. This proves that also *Transformation 2.* is sound.

Since all transformations are sound, algorithm *UnfoldDependencies* is sound and the claim is proven.  $\square$

---

**Theorem 3** Given a source-to-ontology scenario  $\mathcal{M}_{SV} = \{\mathbf{S}, \mathbf{V}, \Sigma_{SV}, \Sigma_V\}$  with view definition  $\Upsilon_V$ , assume  $\Upsilon_V$  conforms to the restrictions in Definition 4. Call  $\mathcal{M}_{ST'}^{emb} = \{\mathbf{S}, \mathbf{T}', \Sigma_{ST'}, \Sigma_{T'}\}$ , the source-to-target rewritten scenario computed by algorithm *UnfoldDependencies*, where  $\mathbf{T}'$  is obtained from  $\mathbf{T}$  by enriching it with a finite set of new relation symbols  $TGD_0, TGD_1, \dots$ . Then  $\mathcal{M}_{ST'}^{emb}$  only contains embedded dependencies (i.e. tgds, egds, and denial constraints).

*Proof:* Assume  $\Upsilon_V$  conforms to Definition 4. We now show that algorithm *UnfoldDependencies* does not introduce any disjunction during the rewriting.

To start, we notice that the original source-to-ontology mapping only contains ordinary embedded dependencies, and therefore no disjunction nor negation is present. Notice also that the premise of source-to-target tgds only contains source symbols, and these are not rewritten.

By looking at algorithm *UnfoldDependencies*, we notice that a disjunction can only be introduced when a dependency  $d : \phi(\bar{x}) \rightarrow \exists \bar{y} : \psi(\bar{x}, \bar{y})$  containing a negated atom  $\neg L(\bar{x})$  in the premise, and a non-empty conclusion, is rewritten to yield  $d' : \phi(\bar{x}) \rightarrow (\exists \bar{y} : \psi(\bar{x}, \bar{y})) \vee L(\bar{x})$ .

957 To see in which cases this may happen, we now want to investigate how  
 958 the negated atom in the premise of  $d$  has appeared in the first place. Recall  
 959 that the original tgds and egds do not contain negations. By reasoning on the  
 960 transformations, we notice that this may happen only in two cases:

- 961 (i) the first case is the one in which  $d$  was originally a denial constraint of the  
 962 form  $d_i : \phi(\bar{x}) \rightarrow \perp$  with two different negated atoms,  $L(\bar{x}), L'(\bar{x})$  in the premise;  
 963 in this case,  $d_i$  is initially rewritten to move  $L'(\bar{x})$  to the conclusion according  
 964 to *Transformation 3.* to yield  $d : \phi'(\bar{x}) \rightarrow L'(\bar{x})$ , and then also  $L(\bar{x})$  according  
 965 to *Transformation 4.*, to yield  $d'$  as discussed above;
- 966 (ii) the second case is the one in which  $d$  was originally an egd of the form  
 967  $d_j : \phi(\bar{x}) \rightarrow x = x'$ , and  $\phi(\bar{x})$  contained a negated atom that is then moved to  
 968 the conclusion by introducing a disjunction.

969 Consider first case (i). Recall that denial constraints are introduced exclu-  
 970 sively by *Transformation 2.* when one of the dependencies has a negated atom  
 971 in its conclusion. Therefore, for case (i) to happen, we need:

- 972 • a tgd with a view symbol  $V$  in its conclusion, that is unfolded according  
 973 to *Transformation 1.* to introduce a negated view atom  $V'(\bar{x}, \bar{y})$ ;
- 974 • atom  $V'(\bar{x}, \bar{y})$  is removed by *Transformation 2.*, to generate a new tgd  $d^1$   
 975 in which it appears positively in the premise;
- 976 • atom  $V'(\bar{x}, \bar{y})$  in the premise of  $d^1$  is again unfolded according to *Tran-*  
 977 *formation 1.*, to introduce two different negated atoms  $\neg L(\bar{x}), \neg L'(\bar{x})$  in the  
 978 premise of  $d^1$ ;
- 979 • these are rewritten according to *Transformation 3.* first, and then *Tran-*  
 980 *formation 4.*, as discussed above, to generate a ded.

981 We notice, however, that this is not possible by Definition 4, since it would  
 982 require a view ( $V$ ), that negatively depends on another ( $V'$ ), and this in turn  
 983 depends on two negated atoms.

984 Let us now consider case (ii) above. This requires that one of the original  
 985 egds contains a view symbol that is unfolded to introduce a negated atom in  
 986 the premise. This is, however, also prevented by Definition 4.

987 This proves that under the restrictions of Definition 4, no disjunction is  
 988 introduced by the algorithm, and therefore the resulting set of dependencies is  
 989 a set of standard embedded dependencies (tgds, egds, and denial constraints).  
 990  $\square$

991 **Appendix B. Complete Rewriting for the Running Example**

Source schema: *S-WorkerGrades*(*WorkerId*, *Year*, *Grade*, *SalaryInc*)  
*S-Stats*(*WorkerId*, *WorkerName*, *MinGrade*, *MaxGrade*)

Target schema: *Employees*(*Id*, *Name*)  
*Evaluations*(*EmployeeId*, *Year*)  
*PositiveEvals*(*EmployeeId*, *Year*, *SalaryInc*)  
*Penalized*(*EmployeeId*, *Year*)  
*Warned*(*EmployeeId*, *Date*)

992

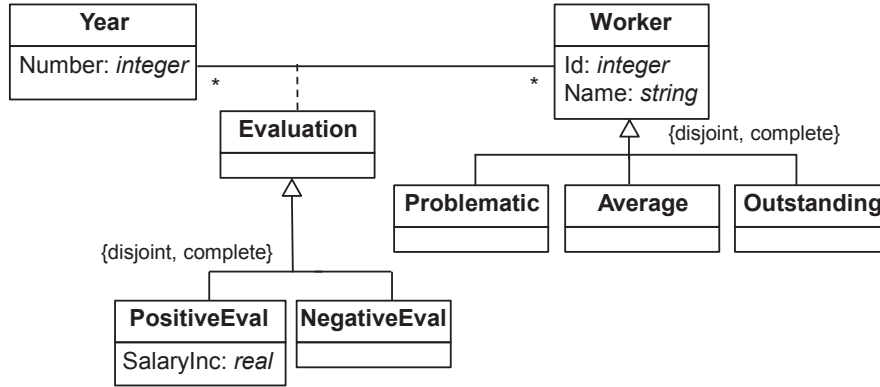


Figure B.7: Target Ontology.

993 **View definitions for the target ontology:**

$Worker(id, name) \Leftarrow Employees(id, name)$   
 $Evaluation(workerId, year) \Leftarrow Evaluations(workerId, year)$   
 $PositiveEval(workerId, year, salaryInc) \Leftarrow Evaluation(workerId, year),$   
 $PositiveEvals(workerId, year, salaryInc)$   
 $NegativeEval(workerId, year) \Leftarrow Evaluation(workerId, year),$   
 $\neg PositiveEval(workerId, year, salaryInc)$   
 $Problematic(id, name) \Leftarrow Worker(id, name), Penalized(id, year)$   
 $Problematic(id, name) \Leftarrow Worker(id, name), \neg PositiveEval(id, year, salaryInc)$   
 $Outstanding(id, name) \Leftarrow Worker(id, name), \neg NegativeEval(id, year), \neg Warned(id, date)$   
 $Average(id, name) \Leftarrow Worker(id, name), \neg Outstanding(id, name), \neg Problematic(id, name)$   
 $Year(number) \Leftarrow Evaluations(employeeId, number)$

994

995 **Source-to-ontology mapping dependencies:**

$m_0 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $maxgr > 4, mingr < 9 \rightarrow Average(id, name), Worker(id, name)$   
 $m_1 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $mingr \geq 9 \rightarrow Outstanding(id, name), Worker(id, name)$   
 $m_2 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $maxgr \leq 4 \rightarrow Problematic(id, name), Worker(id, name)$   
 $m_3 : \forall id, yr, gr, sinc :$   
 $S-WorkerGrades(id, yr, gr, sinc), gr \geq 5 \rightarrow \exists name : PositiveEval(id, yr, sinc),$   
 $Evaluation(id, yr), Worker(id, name), Year(yr)$   
 $m_4 : \forall id, yr, gr, sinc :$   
 $S-WorkerGrades(id, yr, gr, sinc), gr < 5 \rightarrow \exists name : NegativeEval(id, yr),$   
 $Evaluation(id, yr), Worker(id, name), Year(yr)$

996

997

**Ontology egds:**

$e_0 : \forall id, name_1, name_2 : Worker(id, name_1), Worker(id, name_2) \rightarrow name_1 = name_2$   
 $e_1 : \forall id_1, id_2, name : Outstanding(id_1, name), Outstanding(id_2, name) \rightarrow id_1 = id_2$

998

999

**Rewriting of the mapping dependencies into source-to-target:**

$m_0^1 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $maxgr > 4, mingr < 9 \rightarrow Employees(id, name), TGD_1(id, name)$   
 $m_0^2 : \forall id, name : TGD_1(id, name), Employees(id, name) \rightarrow$   
 $\exists yr : (Evaluations(id, yr), TGD_2(id, yr))$   
 $\vee \exists date : Warned(id, date)$   
 $m_0^3 : \forall id, yr : TGD_2(id, yr), Evaluations(id, yr), PositiveEvals(id, yr, sinc) \rightarrow \perp$   
 $m_0^4 : \forall id, name, yr : TGD_1(id, name), Employees(id, name), Penalized(id, yr) \rightarrow \perp$   
 $m_0^5 : \forall id, name : TGD_1(id, name), Employees(id, name) \rightarrow$   
 $\exists yr, sinc : Evaluations(id, yr), PositiveEvals(id, yr, sinc)$   
 $m_1^1 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $mingr \geq 9 \rightarrow Employees(id, name), TGD_3(id)$   
 $m_1^2 : \forall id, yr : TGD_3(id), Penalized(id, yr) \rightarrow \perp$   
 $m_1^3 : \forall id, yr : TGD_3(id), Evaluations(id, yr) \rightarrow \exists sinc : PositiveEvals(id, yr, sinc)$   
 $m_2^1 : \forall id, yr, gr, sinc, name, mingr, maxgr :$   
 $S-WorkerGrades(id, yr, gr, sinc), S-Stats(id, name, mingr, maxgr),$   
 $maxgr \leq 4 \rightarrow \exists yr' : (Employees(id, name), Penalized(id, yr'))$   
 $\vee (Employees(id, name), TGD_4(id))$   
 $m_2^2 : \forall id, yr : TGD_4(id), Evaluations(id, yr), PositiveEvals(id, yr, sinc) \rightarrow \perp$   
 $m_3^1 : \forall id, yr, gr, sinc :$   
 $S-WorkerGrades(id, yr, gr, sinc), gr \geq 5 \rightarrow Evaluations(id, yr),$   
 $PositiveEvals(id, yr, sinc)$   
 $m_4^1 : \forall id, yr, gr, sinc :$   
 $S-WorkerGrades(id, yr, gr, sinc), gr < 5 \rightarrow Evaluations(id, yr), TGD_2(id, yr)$

1000

1001 **Rewriting of the ontology egds into target dependencies:**

$$\begin{aligned}
 e_0^1 &: \forall id, name_1, name_2 : Employees(id, name_1), Worker(id, name_2) \rightarrow name_1 = name_2 \\
 e_1^1 &: \forall id_1, id_2, name : Worker(id_1, name), Worker(id_2, name) \rightarrow id_1 = id_2 \\
 &\quad \vee \exists year : (Evaluations(id_1, year), TGD_5(id_1, year)) \\
 &\quad \vee \exists date' : Warned(id_1, date') \\
 &\quad \vee \exists year : (Evaluations(id_2, year), TGD_5(id_2, year)) \\
 &\quad \vee \exists date' : Warned(id_2, date') \\
 e_1^2 &: \forall id, year : TGD_5(id, year), Evaluations(id, year), \\
 &\quad PositiveEvals(id, year, sinc) \rightarrow \perp
 \end{aligned}$$

1002

1003 The rewriting of mapping dependencies and ontology egds has been simpli-  
 1004 fied (for readability sake): (1) removed redundant atoms, (2) reused relational  
 1005 symbol  $TGD_2$  in  $m_4^1$  (instead of creating a new  $TGD_i$  that would be identical  
 1006 to  $TGD_2$ ), and similarly, (3) used symbol  $TGD_5$  twice in  $e_1^1$ , instead of using  
 1007  $TGD_5$  and another fresh symbol  $TGD_6$ .