Interactive Mapping Specification with Exemplar Tuples

Angela Bonifati, Ugo Comignani, Emmanuel Coquery, Romuald Thion

Interactive Mapping Specification with Exemplar Tuples

While schema mapping specification is a cumbersome task for data curation specialists, it becomes unfeasible for non-expert users, who are unacquainted with the semantics and languages of the involved transformations.

In this paper, we present an interactive framework for schema mapping specification suited for non-expert users. The underlying key intuition is to leverage a few exemplar tuples to infer the underlying mappings and iterate the inference process via simple user interactions under the form of boolean queries on the validity of the initial exemplar tuples. The approaches available so far are mainly assuming pairs of complete universal data examples, which can be solely provided by data curation experts, or are limited to poorly expressive mappings.

We present several exploration strategies of the space of all possible mappings that satisfy arbitrary user exemplar tuples. Along the exploration, we challenge the user to retain the mappings that fit the user's requirements at best and to dynamically prune the exploration space, thus reducing the number of user interactions. We prove that after the refinement process, the obtained mappings are correct. We present an extensive experimental analysis devoted to measure the feasibility of our interactive mapping strategies and the inherent quality of the obtained mappings.

CCS Concepts: • **Information systems** → **Data exchange**;

Additional Key Words and Phrases: data integration; mapping refinement; user interactions

ACM Reference format:
. 2018. Interactive Mapping Specification with Exemplar Tuples. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (May 2018), 38 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Schema mappings [19] are declarative specifications, typically in first-order logic, of the semantic relationship between elements of a source schema and a target schema. They constitute key *data programmability primitives*, leading database users to be empowered with programming facilities on top of large shared databases. Mappings are usually specified and tested in enterprise IT and several other domains by data architects, also known as *developers of engineered mappings* [11]. Several paradigms have been proposed to aid data architects to specify engineered mappings. The first paradigm relies on visual specification of mappings using user-friendly graphical interfaces, as in several mapping designers [10, 30]. Such graphical tools help the data architects design a mapping between schemas in a high-level notation. A major drawback of these approaches is that the generation of mappings in a programming language or in a query language from graphical primitives is dependent of the specific tool. As a consequence, the same graphical specification might be translated into different and incomparable declarative mappings by two different tools, leading to inconsistencies. In order to tackle such impedance mismatch, model management operators have been proposed in [11] to provide a general-purpose mapping designer that can be adapted

0362-5915/2018/5-ART1 $15.00
https://doi.org/0000001.0000001

(i) Source instance E_S^1 :

A (*Airline*):

IdAirline	*Name*	*Town*
a0	AA	L.A.
a1	MAI	Miami

F (*Flight*):

IdFlight	*From*	*To*	*IdAirline*
f0	Miami	L.A.	a0
f1	L.A.	Miami	a1

TA (*TravelAgency*):

IdAgency	*Name*	*Town*
ag0	TC	L.A.

(ii) Canonical mapping:

$F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0)$
$\wedge\ F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_1)$
$\wedge\ A(\text{idAir}_0, \text{name}_1, \text{town}_1)$
$\wedge\ A(\text{idAir}_1, \text{name}_2, \text{town}_2)$
$\wedge\ TA(\text{idAg}, \text{name}_3, \text{town}_1)$
$\rightarrow \exists \text{idC}_0, \text{idC}_1, \text{idC}_2, \text{idF}_2, \text{idF}_3,$
$Co(\text{idC}_2, \text{name}_3, \text{town}_1)$
$\wedge\ Co(\text{idC}_0, \text{name}_1, \text{town}_1)$
$\wedge\ Co(\text{idC}_1, \text{name}_2, \text{town}_2)$
$\wedge\ Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0)$
$\wedge\ Arr(\text{town}_1, \text{idF}_2, \text{idC}_0)$
$\wedge\ Dpt(\text{town}_1, \text{idF}_3, \text{idC}_1)$
$\wedge\ Arr(\text{town}_2, \text{idF}_3, \text{idC}_1)$

(iii) Target instance E_T^1 :

Co (*Company*):

IdCompany	*Name*	*Town*
c0	AA	L.A.
c1	MAI	Miami
c2	TC	L.A.

Dpt (*Departure*):

Town	*IdFlight*	*IdCompany*
Miami	f2	c0
L.A.	f3	c1

Arr (*Arrival*):

Town	*IdFlight*	*IdCompany*
L.A.	f2	c0
Miami	f3	c1

(iv) Source instance E_S^2 :

A (*Airline*):

IdAirline	*Name*	*Town*
a0	AF	Paris

F (*Flight*):

IdFlight	*From*	*To*	*IdAirline*
f0	Lyon	Paris	a0
f1	Paris	Lyon	a1

TA (*TravelAgency*):

IdAgency	*Name*	*Town*
ag0	DT	Paris

Airp (*Airport*):

IdAirport	*Name*	*Town*
ap0	SE	Lyon

(v) Canonical mapping:

$F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0)$
$\wedge\ F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_1)$
$\wedge\ A(\text{idAir}_0, \text{name}_1, \text{town}_1)$
$\wedge\ TA(\text{idAg}, \text{name}_3, \text{town}_1)$
$\wedge\ Airp(\text{idAp}, \text{name}_2, \text{town}_2)$
$\rightarrow \exists \text{idC}_0, \text{idA}, \text{idF}_2,$
$Co(\text{idC}_0, \text{name}_1, \text{town}_1)$
$\wedge\ Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0)$
$\wedge\ Arr(\text{town}_1, \text{idF}_2, \text{idC}_0)$
$\wedge\ Airp'(\text{idA}, \text{name}_2, \text{town}_2)$

(vi) Target instance E_T^2 :

Co (*Company*):

IdCompany	*Name*	*Town*
c0	AF	Paris

Dpt (*Departure*):

Town	*IdFlight*	*IdCompany*
Lyon	f2	c0

Arr (*Arrival*):

Town	*IdFlight*	*IdCompany*
Paris	f3	c0

(vii) Final mapping after refinement:

$$\Sigma_{final} = \{TA(\text{idAg}, \text{name}_3, \text{town}_1) \rightarrow \exists \text{idC}_2, Co(\text{idC}_2, \text{name}_3, \text{town}_1);$$
$$F(\text{idF}_0, \text{town}_2, \text{town}_1', \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1'')$$
$$\rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1', \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1'')\}$$

Fig. 1. Running example: exemplar tuples (E_S^1, E_T^1) and (E_S^2, E_T^2) (i), (iii), (iv) and (vi), resp.; Canonical mapping (ii) and (v), and Final mapping (vii).

to a wide variety of tools for data programmability. Model management, however, is also suited for expert users. The third paradigm is to generate the desired mappings from representative data examples [4, 5, 24], i.e., a pair of source and target instances, provided by the expert user. However, such data examples are assumed to be solutions of the mapping at hand and representative of all other solutions. Notwithstanding the progress made in mapping specification thanks to the aforementioned approaches, all the above paradigms have in common the fact that they are intended for expert users. Such users are typically acquainted with mapping specification tools and possess complete knowledge of the mapping domains, the formal semantics of mappings and their solution. Ultimately, they are capable of formulating queries or writing customized code.

As also observed in [11], at the other end of the spectrum lies end-users, who find relationships between data and build mapping examples as they go, as in mining heterogeneous data sources, web search, scientific and personal data management. More and more ordinary users are in fact confronted on a daily basis with user-driven data exploration scenarios, such as those exposed by dataspaces [20]. As a consequence, the problem of mapping specification for such classes of users is even more compelling.

To tackle the above problem, in this paper we set forth a novel approach for *Interactive Mapping Specification* (IMS) that bootstraps with *exemplar tuples*, corresponding to a limited number of

tuples provided by non-expert users. Such tuples are employed to challenge the user with simple boolean questions, which are intended to drive the inference process of the mapping that the user has in mind and that is unknown beforehand.

(IMS) Given exemplar tuples as input pairs $\{(E_S^1, E_T^1); \ldots; (E_S^n, E_T^n)\}$ *provided by a non-expert user and a mapping* $\mathcal{M}$ *that the user has in mind, the* Interactive Mapping Specification *problem is to discover, by means of boolean interactions, a mapping* $\mathcal{M}'$ *such that each* $(E_S^i, E_T^i) \in \{(E_S^1, E_T^1); \ldots; (E_S^n, E_T^n)\}$ *satisfy* $\mathcal{M}'$ *and* $\mathcal{M}'$ *generalizes* $\mathcal{M}$.

Notice that the user-provided exemplar tuples may turn to be not well chosen or even ambiguous with respect to the mapping $\mathcal{M}$ that the user has in mind. Moreover, exemplar tuples are not supposed to be solutions nor universal solutions of the mapping that needs to be inferred. Whereas a wealth of research on schema mapping understanding and refinement has been conducted in databases [4, 17, 21, 22, 33] since the pioneering work of Clio [28], these approaches assume more sophisticated input (such as an initial mapping to refine and the schemas and schema constraints) and/or more complex user interactions. Although exemplar tuples reminisce data examples [5], they are fundamentally different in that they are not meant to be universal. Furthermore, the mappings we consider in this paper are unrestricted GLAV mappings. We present a detailed comparison with previous work in Section 7, and a comparative analysis with [6, 13] in Section 6.

Query specification has been recognized as challenging for non-expert users and more time-consuming than executing the query itself [25]. We argue that mapping specification is even more arduous for such users, merely because mappings embody semantic relationships between inherently complex queries. Despite many recent efforts on query specification for non-expert users [1, 2, 12, 18, 27], these works are not applicable to mapping specification for non-expert users, which we address in this paper (for more details, we refer the reader to Section 7).

Figure 1 illustrates our running scenario, where a non-expert user needs to establish a mapping between two databases exhibiting travel information. The source database schemas are made of four relations, *Airline*, *Flight*, *TravelAgency*, and *Airport* (abbreviated respectively as *A*, *F*, *TA* and *Airp*). The target database schemas contains four relations *Company*, *Departure* and *Arrival* (resp. *Co*, *Dpt* and *Arr*).

In this scenario, user provide two pairs of source and target exemplar tuples sets : (E_S^1, E_T^1) and (E_S^2, E_T^2). For each of this pairs, source and target databases are reported in the left-hand and right-hand sides of the Figure 1, respectively. We can observe that the number of tuples per each table is small: the user is not intended to provide a complete instance but only a small set of representative tuples. We can also easily identify a few inherent ambiguities within the provided exemplar tuples. For instance, in (E_S^1, E_T^1), the constant `L.A.` represents both the town where the travel agency is located (in relation *TA*, which contains travel agencies information) and the destination of a flight (in the corresponding relation *F*). If we would consider these exemplar tuples as the ground truth, we would translate them into *canonical mapping* illustrated in Figure 1 (ii) and (v). Such mappings, however, *reflects the ambiguities* of the provided exemplar tuples, by assuming that *all solutions* must have two airline and that the travel agency (resp. the airport) must be located in the same city than an airline headquarters. Thus, from a logical viewpoint, such mappings are way *too specific*. Moreover, such mappings can be quite large and unreadable in real-world scenarios, as they embeds all the exemplar tuples altogether. Our *mapping specification process* builds upon end-user exemplar tuples, which can be ambiguous and ill-defined. Hence, it aims at deriving smaller refined and normalized mappings through simple user interactions, in order to obtain more controllable mappings closer to what the user has in mind (illustrated in Figure 1 (iv)). The rest of the paper is devoted to explain such a transformation.

The main contributions of our paper are summarized as follows:

- We define a mapping specification process for non-expert users that bootstraps with exemplar tuples, and works for general GLAV mappings. The user is challenged with boolean questions over even smaller refinement-driven tuples generated from the initial exemplar tuples. The space of possible solutions is represented as a quasi-lattice, on top of which a dynamic pruning keeps the number of user interactions reasonably low. The introduction of quasi-lattices, instead of separate upper semi-lattices as used in [13], allows to avoid redundant explorations and leads to reduce the number of required user interactions.
- We prove that the generated mappings have *irreducible right-hand sides*. Combined with redundant mapping elimination, this guarantees that the obtained refined mappings are in normal form [23]. Intuitively, normalized mapping are more self-explanatory and understandable for end-users compared to monolithic canonical mappings.
- We prove that the refinement process always produces *a more general mapping* than the canonical mapping and is always implied by the mapping expected by the user. As an example, an illustration of the obtained mapping for our running example is in Figure 1 (iv), which can be confronted with the canonical mappings of Figure 1 (ii) and (iv). We define the condition under which our system will produce a mapping logically equivalent to the one expected by the user. This is a major improvement of the work done by [13], as they provide less formal guarantees about the produced mapping.
- We introduce the adoption of integrity constraints (ICs) in order to reduce the number of asked questions, when the approach in [13] does not allow the use of such constraints. To this end, we present a modified version of the problem statement, namely IMS_{IC} and we study the various classes of allowed ICs.
- We experimentally gauge diminution of the number of asked questions induced by the introduction of quasi-lattices compared to the work in [13]. Moreover, we experimentally gauge the effectiveness of our approach, by comparing the sizes of exemplar tuples with the size of universal solutions.

The rest of this paper is organized as follows. Section 2 introduces the notation used in the rest of our paper. Specific background on the mapping generation from exemplar tuples is detailed in Section 3.1. The bulk of our approach is described in Section 3.2. A formal general framework is described in Section 4, as well as proofs of correctness and completeness of this framework and the implementation detailled in previous section. Formal considerations about the use of integrity constraints in order to reduce the number of questions is presened in Section 5 and an extensive experimental study is presented in Section 6. Related work is devoted to Section 7. We conclude the paper in Section 8.

2 PRELIMINARIES

We briefly introduce various concepts from the data exchange framework [19] that we use in this paper. Given two disjoint countably infinite sets of constants C and variables $\mathcal{V}$, we assume a bijective function $\bar{\theta}$, such that if $\bar{\theta}(\mathsf{x}_i) = \mathsf{c}_i$, then $\mathsf{c}_i \in C$ is the constant associated to the variable $\mathsf{x}_i \in \mathcal{V}$ and $\bar{\theta}^{-1}(\mathsf{c}) = \mathsf{x}$. A *tuple* over a relation R has the form $R(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ where $\mathsf{c}_i \in C$, while an *atom* has the form $R(\mathsf{x}_1, \ldots, \mathsf{x}_n)$ where $\mathsf{x}_i \in \mathcal{V}$. The bijection $\bar{\theta}$ naturally extends to a bijection between (conjunctions of) atoms and (sets of) tuples.

A *(schema) mapping* is a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema disjoint from $\mathbf{S}$, and Σ is a set of *tuple-generating dependency* (tgd for short) over schemas $\mathbf{S}$ and $\mathbf{T}$. A tgd is a first-order logical formula the form $\phi(\overline{x}) \rightarrow \exists \overline{y}, \psi(\overline{x}, \overline{y})$ where $\overline{x}$ and $\overline{y}$ are vectors of variables, $\overline{x}$ being universally quantified, and where both ϕ and ψ are conjunctions of atoms. In

this paper, we only consider source-to-target tgd (s-t tgds for short), in which atoms in ϕ are over relations in $\mathbf{S}$ and atoms in ψ are over relation in $\mathbf{T}$. We consider *GLAV* mappings where a tgd can contain more than one atom in ϕ and in ψ.

Two tgds $\sigma_1 : \phi_1(\overline{x}_1) \rightarrow \exists\overline{y}_1, \psi_1(\overline{x}_1, \overline{y}_1)$ and $\sigma_2 : \phi_2(\overline{x}_2) \rightarrow \exists\overline{y}_2, \psi_2(\overline{x}_2, \overline{y}_2)$ are ψ-equivalent if there exists a morphism $\mu : \psi_2(\overline{x}_2, \overline{y}_2) \rightarrow \psi_1(\overline{x}_1, \overline{y}_1)$ such that $\psi_1 \equiv \mu(\psi_2)$, and μ match existential variables in $\overline{y}_2$ only with existential variables in $\overline{y}_1$ and universally quantified variables in $\overline{x}_2$ are matched only with universal variables in $\overline{x}_1$. We denote ψ-equivalence between two tgds σ_1 and σ_2 by the following notation: $\sigma_1 \equiv_\psi \sigma_2$.

Analogously, two tgds $\sigma_1 : \phi_1(\overline{x}_1) \rightarrow \exists\overline{y}_1, \psi_1(\overline{x}_1, \overline{y}_1)$ and $\sigma_2 : \phi_2(\overline{x}_2) \rightarrow \exists\overline{y}_2, \psi_2(\overline{x}_2, \overline{y}_2)$ are ϕ-equivalent if there exists a morphism $\mu : \phi_2(\overline{x}_2) \rightarrow \phi_1(\overline{x}_1)$ such that $\phi_1 \equiv \mu(\phi_2)$. We denote ϕ-equivalence between two tgds σ_1 and σ_2 by the following notation: $\sigma_1 \equiv_\phi \sigma_2$.

An instance E_T over $\mathbf{T}$ is a *solution* for a source instance E_S over $\mathbf{S}$ under a mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ iff $(E_S, E_T) \models \Sigma$. A mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ *logically entails* a mapping $\mathcal{M}' = (\mathbf{S}, \mathbf{T}, \Sigma')$, denoted by $\mathcal{M} \models \mathcal{M}'$, if for every (E_S, E_T) if $(E_S, E_T) \models \Sigma$ then $(E_S, E_T) \models \Sigma'$. Two mappings $\mathcal{M}$ and $\mathcal{M}'$ are *logically equivalent*, denoted by $\mathcal{M} \equiv_l \mathcal{M}'$, if $\mathcal{M} \models \mathcal{M}'$ and $\mathcal{M}' \models \mathcal{M}$. When comparing mappings, we say that $\mathcal{M}$ is *more general* than $\mathcal{M}'$ if $\mathcal{M} \models \mathcal{M}'$. Informally, this means that tgds in $\mathcal{M}$ are triggered more often than those in $\mathcal{M}'$.

Let E_S and $E_S{}'$ be two instances over the same schema. A *homomorphism* from E_S to E_S is a function h from constants in E_S to constants in $E_S{}'$ such that for any tuple $R(c_1, \ldots, c_n)$ in the instance E_S, the tuple $R(h(c_1), \ldots, h(c_n))$ belongs to E_S. An instance E_T is an *universal solution* for the instance E_S under a mapping $\mathcal{M}$ if E_T is a solution for E_S and if for each solution $E_T{}'$ for E_S under $\mathcal{M}$, there exists a homomorphism $h : E_T \rightarrow E_T{}'$ such that $h(\mathsf{c}) = \mathsf{c}$ for every constant c appearing both in E_T and $E_T{}'$.

It was shown in [19] that the result of chasing E_S with Σ is a universal solution. The application of the *chase procedure*, denoted by $\textsc{chase}(\Sigma, E_S)$, is as follows: for each tgd $\phi(\overline{x}) \rightarrow \exists\overline{y}, \psi(\overline{x}, \overline{y}) \in \Sigma$, if there exists a substitution μ of $\overline{x}$ such that all atoms in $\phi(\overline{x})$ can be mapped to tuples in E_S, extend this substitution to μ' by picking a fresh new constant for each variable in $\overline{y}$ and finally add all atoms of $\psi(\overline{x}, \overline{y})$ instantiated to tuples with μ' into E_T. Another key result of the literature that we use in this paper is borrowed from [9] and states that $\Sigma \models \phi(\overline{x}) \rightarrow \exists\overline{y}, \psi(\overline{x}, \overline{y})$ *if and only if* there exists a substitution μ' extending an arbitrary μ such that $\mu'(\psi(\overline{x}, \overline{y})) \subseteq \textsc{chase}(\Sigma, \mu(\phi(\overline{x})))$.

The chase procedure give us a way to test the logical implication of mappings by the use of the following property : $\mathcal{M} \models \mathcal{M}'$ *if and only if* $\forall\sigma' \in \Sigma', \psi_{\sigma'} \subseteq \textsc{chase}(\Sigma, \phi_{\sigma'})$[26].

Finally, for the schema mapping normalization, we borrow two notions from [23]: *split-reduced* mappings and *σ-redundant* mappings. While *split-reduction* breaks a tgd into a logically equivalent set of tgds with right-hand sides having non overlapping existentially quantified variables, *σ-redundancy* encodes the presence of unnecessary tgds. We report formal definitions below. Let $\sigma : \phi(\overline{x}) \rightarrow \exists\overline{y}, \psi(\overline{x}, \overline{y})$ be a tgd. We say that σ is *split-reduced* if there is no pair of tgds $\sigma_1 : \phi_1(\overline{x}) \rightarrow \exists\overline{y}_1, \psi_1(\overline{x}, \overline{y}_1)$ and $\sigma_2 : \phi_2(\overline{x}) \rightarrow \exists\overline{y}_2, \psi(\overline{x}, \overline{y}_2)$ such that $\overline{y}_1 \cap \overline{y}_2 = \emptyset$ and $\{\sigma\} \equiv_l \{\sigma_1; \sigma_2\}$. A mapping $(\mathbf{S}, \mathbf{T}, \Sigma)$ is *split-reduced* if, for all tgd $\sigma \in \Sigma$, σ is *split-reduced*. According to [23], given a mapping $\mathcal{M}$, it is always possible to find a split-reduced mapping $\mathcal{M}'$ that is equivalent to $\mathcal{M}$. Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ be a schema mapping and $\sigma \in \Sigma$ a tgd. We say that $\mathcal{M}$ is *σ-redundant*, w.r.t. logical equivalence, iff $\Sigma \setminus \{\sigma\} \equiv_l \Sigma$. Such equivalence can be tested using the *chase procedure* as a proof procedure for the implication problem by checking whether $\Sigma \setminus \{\sigma\} \models \sigma$.

We briefly recall a few notions on partitions. A partition of a set $\mathcal{W}$ is a set $\mathcal{P}$ of disjoint and non-empty subsets of $\mathcal{V}$ called *blocks*, such that $\bigcup_{b \in \mathcal{P}} b = \mathcal{W}$. The set of all partitions of $\mathcal{W}$ is denoted by Part($\mathcal{W}$). Two objects of $\mathcal{W}$ that are in the same block of a partition $\mathcal{P}$ are denoted by

$a \equiv_{\mathcal{P}} b$. The set of all partitions of $\mathcal{W}$ form a complete lattice under the partial order :

$$\mathcal{P}_0 \leq \mathcal{P}_1 \Leftrightarrow \forall x, y \in \mathcal{W}, \left(x \equiv_{\mathcal{P}_0} y \Rightarrow x \equiv_{\mathcal{P}_1} y\right)$$

This partial order formally captures the intuitive notion of refinement of a partition.

3 MAPPING REFINEMENT

In this section, we describe the key components of our interactive mapping specification process, as depicted in Figure 2.

3.1 Exemplar tuples and mappings

Exemplar tuples are defined as a pair of source and target instances (E_S, E_T). Tuples in these two instances can be arbitrary chosen in the sense that there's no need for him to provide an instance E_T which is an universal solution to E_S through his expected mapping. Instead, given the source instance E_S, the user can populate the target instance E_T with only few tuples coming from the universal solution to E_S through his expected mapping. In other words, The formal definition of such a pair of instance is given in the following definition :

Definition 3.1 (Exemplar tuples). Let Σ_{exp} be a mapping. Then an *exemplar tuple* for Σ_{exp} is a pair of instances (E_S, E_T) such that :

$$E_T \subseteq \textsc{chase}(\Sigma_{exp}, E_S)$$

Henceforth, they are simply called *exemplar tuples* whenever Σ_{exp} is clear from the context. A set of exemplar tuples is denoted by the letter $\mathcal{E}$.

Example 3.2. Given a mapping $\Sigma = \{S(\mathrm{x}, \mathrm{y}) \rightarrow \exists z, T(\mathrm{x}, \mathrm{z}); S'(x, y) \rightarrow T(\mathrm{x}, \mathrm{z}) \wedge T'(\mathrm{z}, \mathrm{y})\}$. Given a source instance $I = \{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d}); S'(\mathsf{e}, \mathsf{f})\}$. Then, a possible exemplar tuple can be the pair (E_S, E_T) with $E_S = I$ and :

$$E_T \subseteq \{T(\mathsf{a}, \mathsf{n}_1); T(\mathsf{c}, \mathsf{n}_2); T(\mathsf{e}, \mathsf{n}_3); T'(\mathsf{n}_3, \mathsf{f})\}$$

An other exemplar tuple can be the pair $(\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}, \{T(\mathsf{a}, \mathsf{n}_1)\})$, which exemplifies the tgd $S(\mathrm{x}, \mathrm{y}) \rightarrow \exists z, T(\mathrm{x}, \mathrm{z})$ of Σ.

A counter example is the pair $(\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}, \{T(\mathsf{a}, \mathsf{n}_1); T'(\mathsf{n}_2, \mathsf{b})\})$. Here, the tgd $S(\mathrm{x}, \mathrm{y}) \rightarrow \exists z, T(\mathrm{x}, \mathrm{z})$ is exemplified by the tuples $S(\mathsf{a}, \mathsf{b})$ and $T(\mathsf{a}, \mathsf{n}_1)$, but this pair is not an exemplar tuples because the tuple $T'(\mathsf{n}_2, \mathsf{b})$ cannot be deduced from Σ with the source tuples $\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}$, and thus is not consistent with our definition 3.1.

Given a set of *exemplar tuples* $\mathcal{E}$, this set is said to be *fully-informative* for a mapping Σ_{exp} if it respect the following definition :

Definition 3.3 (Fully-informative exemplar tuples set). Let Σ_{exp} be a mapping in normal form. Then a *fully-informative exemplar tuples set* for Σ_{exp} is a set of exemplar tuples $\mathcal{E}$ such that each connected component of the tgd in Σ_{exp} is exemplified at least once, i.e. :

$$\forall \sigma \in \Sigma_{exp}, \exists (E_S, E_T) \in \mathcal{E}, \exists E'_S \subseteq E_S \text{ s.t. } (\textsc{chase}(\sigma, E'_S) \neq \emptyset) \wedge (\textsc{chase}(\sigma, E'_S) \subseteq E_T)$$

This definition capture the need of having sufficient information conveyed by the set of *exemplar tuples* provided by the user, in order to allow to retrieve the mapping the user as in mind (or a logically equivalent mapping). This will be proved in Section 4.

Example 3.4. We reuse the sets Σ and I of the previous example 3.2. As stated in example 3.2, the pair $(\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}, \{T(\mathsf{a}, \mathsf{n}_1)\})$ is an exemplar tuple for Σ. But this example does not exemplify the tgd $S'(x, y) \rightarrow T(\mathrm{x}, \mathrm{z}) \wedge T'(\mathrm{z}, \mathrm{y})$, so it violates definition 3.3.

In the following set, a pair for the tgd $S'(x, y) \rightarrow T(\mathrm{x}, \mathrm{z}) \wedge T'(\mathrm{z}, \mathrm{y})$ is added, leading to a fully-informative exemplar tuples sets for Σ :

$$\{(\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}, \{T(\mathsf{a}, \mathsf{n}_1)\});$$
$$(\{S'(\mathsf{e}, \mathsf{f})\}, \{T(\mathsf{e}, \mathsf{n}_3); T'(\mathsf{n}_3, \mathsf{f})\})\}$$

By opposite, the set :

$$\{(\{S(\mathsf{a}, \mathsf{b}); S(\mathsf{c}, \mathsf{d})\}, \{T(\mathsf{a}, \mathsf{n}_1)\});$$
$$(\{S'(\mathsf{e}, \mathsf{f})\}, \{T(\mathsf{e}, \mathsf{n}_3)\})\}$$

is not fully-informative for Σ as there is no exemplar tuple that exemplifies the connected component of tgd $S'(x, y) \rightarrow T(\mathrm{x}, \mathrm{z}) \wedge T'(\mathrm{z}, \mathrm{y})$.

Given an input pair (E_S, E_T), we build a *canonical mapping* as follows. More precisely, given a pair (E_S, E_T), the canonical mapping associated to (E_S, E_T) is the tgd $\phi \rightarrow \psi$ where $\phi = \bar{\theta}^{-1}(E_S)$ and $\psi = \bar{\theta}^{-1}(E_T)$. Informally, the left-hand side ϕ is constructed from E_S by replacing all tuples in E_S by their atoms counterparts, with the constants being replaced by variables. The right-hand side of the canonical mapping is obtained in a similar fashion.

Example 3.5. The canonical mappings corresponding to the exemplar tuples of Figure 1 are represented in Figure 1 (ii) and (v).

However, notice that the canonical mappings of Example 3.5 are extremely rigid. For instance, in the canonical mapping (ii) we can observe that tuples in the source relation *TA* are mandatorily needed in order to obtain tuples in the target relation *Arr*.

This is due to the fact that a canonical mapping is the most specific mapping obtained from the exemplar tuples: it contains *all* the atoms corresponding to E_S on its left-hand side. Since exemplar tuples are not universal by definition [1], this mapping are far too constrained. The envisioned workaround is to refine the canonical mappings into a less constrained one by leveraging simple user interactions.

Intuitively, the refinement of the canonical mappings is done through the following steps: the first is a pre-processing that leads to a single normalized mapping, in which each large tgd of a canonical mappings is divided into *equivalent* set of smaller ones; the second and the third steps revolve around mapping refinement via user interactions that lets simplify the left-hand sides of the tgds. We devote the rest of this subsection to the first step, while we describe the latter steps in the next subsections.

We define formal criteria that capture the quality of a mapping $\mathcal{M}$ intuitively as follows: each tgd in Σ should have a minimal right-hand side and there should be no spurious tgd in Σ. To that purpose, we rely on the two previously introduced notions, i.e. *split-reduced* mappings and σ-redundant mappings[23]. The splitting of the original mapping into smaller tgds turns out to be convenient for mapping refinement, in that it lets the user focus only on the *necessary* atoms implied in the left-hand sides of each reduced tgd. However, as a side effect of split-reduction, we may get redundant tgds in the set Σ. Such redundant tgds are unnecessary and need to be removed to avoid inquiring the user about useless mappings. Finally, we say that $(\mathbf{S}, \mathbf{T}, \Sigma)$ is *normalized* when each tgd in Σ is *split-reduced* and there is no *σ-redundant* tgd in Σ.

[1] If exemplar tuples (E_S, E_T) were *universal*, then (E_S, E_T) $\models \sigma$ where σ is the canonical mapping associated to (E_S, E_T).

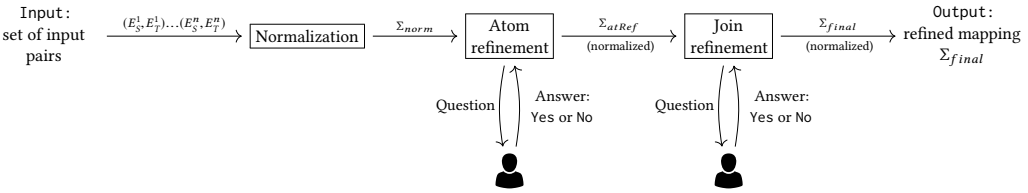


Fig. 2. Interactive mapping specification process.

Example 3.6. The *split-reduction* on the canonical mappings of Figure 1(ii) and (v) leads to the following set of tgds $\Sigma_{splitReduced}$:

$$\begin{aligned}\phi_1 =& F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0) \wedge F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_1) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1)\\ & \wedge A(\text{idAir}_1, \text{name}_2, \text{town}_2) \wedge TA(\text{idAg}, \text{name}_3, \text{town}_1)\\ \phi_2 =& F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0) \wedge F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}') \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1)\\ & \wedge Airp(\text{idAp}, \text{name}_2, \text{town}_2) \wedge TA(\text{idAg}, \text{name}_3, \text{town}_1)\end{aligned}$$

$\Sigma_{splitReduced} = \{$

$\phi_1 \rightarrow \exists \text{idC}_2, Co(\text{idC}_2, \text{name}_3, \text{town}_1);$ (1)

$\phi_1 \rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1);$ (2)

$\phi_1 \rightarrow \exists \text{idC}_1, \text{idF}_3, Dpt(\text{town}_1, \text{idF}_3, \text{idC}_1) \wedge Arr(\text{town}_2, \text{idF}_3, \text{idC}_1) \wedge Co(\text{idC}_1, \text{name}_1, \text{town}_2);$ (3)

$\phi_2 \rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1)\}$ (4)

The *σ-redundancy suppression* on $\Sigma_{splitReduced}$ allow to suppress the redundant tgd (3), which is logically equivalent to tgd (2). The *σ-redundancy suppression* cannot be applied to tgds (2) and (4) as their left-hand sides are different.

This lead to the normalized mapping Σ_{norm} :

$\Sigma_{norm} = \{$

$\phi_1 \rightarrow \exists \text{idC}_2, Co(\text{idC}_2, \text{name}_3, \text{town}_1);$ (1)

$\phi_1 \rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1);$ (2)

$\phi_2 \rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1)\}$ (4)

3.2 Refinement of mappings

The previous section has defined the pre-processing step that leads to a normalized canonical mapping. We now introduce the two refinement steps that constitute the core of our proposal. The assumption underlying our approach is that a non-expert user provides a set of exemplar tuples $\mathcal{E}$ in input and, during the mapping refinement steps, this user will interacts with our system via simple boolean questions about the validity of small data examples. If the provided set of exemplar tuples is a *fully informative* set, then the output mapping is guaranteed to be equivalent to the mapping expected by the user, as it will be proved in Section 4.

In this paper, we assume that the questions about the validity of this data examples are answered by an oracle. This oracle answer question using the following procedure :

*Definition 3.7 (*oracle *answering procedure.).* Let $\mathcal{M}_{exp} = \langle S, T, \Sigma_{exp} \rangle$ be the mapping expected by the *oracle*. Let $(\phi \rightarrow \psi)$ be a tgd.
Then, the *oracle* answer `true` to the question *"Are the tuples $\bar{\theta}(\phi)$ enough to produce $\bar{\theta}(\psi)$?"* if :

$$\psi_\sigma \subseteq \textsc{chase}(\phi_\sigma, \Sigma_{exp})$$

The choice of such a modelisation of users is motivated by the intuition that even if a non-expert user is not able to express the mapping he expects with a logical language, he can rely on domain knowledges to answer if the information contained in a set of source tuples is sufficient to infer a given set of target tuples. In this paper, we don't consider other kinds of users that correspond to a relaxation of the previous conditions and we leave this part to future investigation.

In the rest of this section, for ease of exposition, we assume that the user provides only two pairs (E_S^1, E_T^1) and (E_S^2, E_T^2) of exemplar tuples. However, in practice, the user might provide a larger *set* of exemplar tuples.

For a given input pair (E_S^k, E_T^k), the number of mappings satisfying it may be quite large. Therefore, it is important to provide efficient exploration strategies of the space of mappings in order to reduce the number of questions to ask to the user. An important method used here relies on the fact that we can partition the normalized canonical mapping obtained from user's exemplar tuples in blocks of ψ-equivalent tgds. These sets of tgds are handled together to find morphisms between subsets of their left-hand sides. Such morphisms corresponds to equivalent tgds extracted from different exemplar tuples, so we need to avoid exploring them more than once to reduce the size of the explored space. This is a major difference with the refinement presented in [13], in which each exemplar tuple is explored separately, leading to redundant superfluous interactions with the user.

Two successive steps are applied during refinement: the *atom refinement* step and the *join refinement* step. We illustrate such steps in Figure 2, along with the corresponding user interactions required to obtain the final result, i.e., the refined tgds that meet the user's requirements. The atom refinement step aims at removing unnecessary atoms in the left-hand side of the tgds within the normalized mapping obtained in the pre-processing. The join refinement step applies the removal of unnecessary joins between atoms in each tgd as output by the previous step. During both steps, the user is challenged with specific questions devoted to address ambiguities of the provided exemplar tuples and refine the normalized canonical mapping obtained in the pre-processing step. We focus on the first step in Section 3.3 and we postpone the description of the second step to Section 3.4.

In our approach, we use universally quantified variables as the targets of the refinement algorithms and assume that the existential variables in the right-hand side of the tgds are unambiguous (and appear as such in the input exemplar tuples). In other words, value invention (e.g., the production of labeled nulls in `SQL`) in the target exemplar tuples is supposed to be correct and the user is not inquired about them. This also implies that our algorithms *do not create fresh existential variables* in the tgds. The introduction of such variables would drastically increase the number of mappings to explore and their coverage would entail non-trivial extension of our algorithms, which are beyond the scope of this paper.

3.3 Atom refinement

As discussed in Section 3.1, the normalization produces a *split-reduced* mapping from the canonical mapping in which each tgd has a large left-hand side ϕ. However, some atoms in ϕ may be irrelevant, preventing the triggering of a tgd and causing further ambiguities. To alleviate these ambiguities, Algorithm 1 applies atom refinement on each block of the partition of ψ-equivalent tgds. In the following, we explain its key components and properties.

Algorithm 1 TgdsAtomRefinement(Σ)

Input: A set of tgds Σ to be atom refined.
Output: A set of tgds Σ' where each tgd is atom refined.

```
1: P_Σ ← generate partition of ψ-equivalent tgds from Σ
2: Σ' ← ∅
3: for all b ∈ P_Σ do
4:     let b be ψ-equivalent over ψ_b
5:     C_cand ← generate set of possibles left-hand side candidates from b
6:     C_valid ← generate the upper bound of the quasi-lattice over b
7:     C_invalid ← ∅
8:     while C_cand ≠ ∅ do
9:         e ← SelectAtomSet(C_cand, C_valid)
10:        if AskAtomSetValidity(e, ψ_b) then
11:            add e to C_valid
12:            remove supersets of e from C_valid
13:            remove e and its supersets from C_cand
14:        else
15:            add e to C_invalid
16:            remove e and its subsets from C_cand
17:        end if
18:    end while
19:    for all e ∈ C_valid do
20:        add the tgd (e → ψ) to Σ'
21:    end for
22: end for
23: return Σ'
```

3.3.1 Groups of ψ-equivalent tgds. The first step of atom refinement aims at grouping ψ-equivalent tgds together in order to allow a more efficient exploration of the search space. To this purpose, given $\Sigma_{norm} = \{\sigma_1 \dots \sigma_n\}$ the set of tgds generated during normalisation, we create a partition $\mathcal{P}_{norm}$ of Σ_{norm} in which each block is constituted by ψ-equivalent tgds. More formally, we produce the partition $\mathcal{P}_{norm}$ such that:

$$\forall \sigma_i, \sigma_j \in \Sigma_{norm}, \sigma_i \equiv_{\mathcal{P}_{norm}} \sigma_j \Leftrightarrow \sigma_i \equiv_{\psi} \sigma_j$$

3.3.2 Quasi-lattice for Atom Refinement. The baseline structure for the atom refinement of a block $\mathcal{B} \in \mathcal{P}_{norm}$ is a quasi-lattice. A quasi-lattice is a restriction of a complete lattice to a subset of its nodes, included between an upper and a lower bound.

In our setting, given $\{\phi_1; \dots; \phi_n\}$ the set of the left-hand parts of the tgds in $\mathcal{B}$, the quasi-lattice is build over the complete lattice $\mathcal{L} = (\text{Pow}(\bigcup_{i=1}^{n} \phi^i), \subseteq_h)$ where $\text{Pow}(\bigcup_{i=1}^{n} \phi^i)$ is the powerset of the set of all atoms in the left-hand sides of the tgds in $\mathcal{B}$. For all elements e_x and e_y of $\text{Pow}(\bigcup_{i=1}^{n} \phi^i)$ the least-upper-bound of the set $\{e_x, e_y\}$ is their union.

As atom refinement does not add new constraints in the tgds, we does not create conjunctions which are not subsets of the left-hand side of at least one tgds in $\mathcal{B}$. Thus we can define the *upper bound* of the quasi-lattice as the set $\{\text{At}(\phi_1); \dots; \text{At}(\phi_n)\}$ where $\text{At}(\phi_i)$ is the set of atoms in the conjunction ϕ_i.

Example 3.8. Considering the tgds in Σ_{norm} of Example 3.6, the left-hand sides are made of conjunction ϕ_1 and ϕ_2. The elements of the quasi-lattices we consider are the subsets of the two following sets of atoms (respectively, the sets of atoms in ϕ_1 and ϕ_2) :

$$\{F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0); F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_1);$$
$$A(\text{idAir}_0, \text{name}_1, \text{town}_1); A(\text{idAir}_1, \text{name}_2, \text{town}_2); TA(\text{idAg}, \text{name}_3, \text{town}_1)\}$$
$$\{F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0); F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_1);$$
$$A(\text{idAir}_0, \text{name}_1, \text{town}_1); Airp(\text{idAp}, \text{name}_2, \text{town}_2); TA(\text{idAg}, \text{name}_3, \text{town}_1)\}$$

It is worth to note that many of the subsets of this two sets are homomorphically equivalents. Such an equivalence can be used to leverage common parts of the tgds.

Recalling that our system does not create new existentially quantified variables in the tgds, we need to prune each sets of atoms leading to violate this rule. An existential variables in a tgd correspond to a variable occurring only in the right-hand side, i.e., a variable leading to the creation of new value in the target instance. So, each candidate left-hand side conjunction that does not contain the whole set of right-hand side universal variables will be excluded from the set of candidates. Thus, the set of smallest left-hand side conjunctions containing, at least, all the universal variables of the right-hand side conjunction define the *lower bound* of our quasi-lattice. This restriction takes effect in line 6 of Algorithm 1.

Example 3.9. We illustrate the atom refinement on Example 3.6. As the process stay analogous for each tgd in Σ_{norm}, we focus on the tgds (2) and (4) which right-hand side is:

$$\phi_1 \equiv_h \phi_2 \equiv_h \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1)$$

The set of universally quantified variables in this conjunction is $\{\text{town}_2, \text{town}_1, \text{name}_1\}$. A refined tgd needs to contain at least these variables in its left-hand side. The smallest subsets of the set of atoms given in Example 3.8 for which this assumption is valid are:

$$\{F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0); A(\text{idAir}_0, \text{name}_1, \text{town}_1)\},$$
$$\{F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_0); A(\text{idAir}_0, \text{name}_1, \text{town}_1)\} \text{ and}$$
$$\{A(\text{idAir}_0, \text{name}_1, \text{town}_1); A(\text{idAir}_1, \text{name}_2, \text{town}_2))\}$$

for tgd (2), and :

$$\{F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0); A(\text{idAir}_0, \text{name}_1, \text{town}_1)\},$$
$$\{F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}_0); A(\text{idAir}_0, \text{name}_1, \text{town}_1)\} \text{ and}$$
$$\{A(\text{idAir}_0, \text{name}_1, \text{town}_1); Airp(\text{idAp}, \text{name}_2, \text{town}_2)\}$$

for tgd (4). This sets constitute the lower bound of our quasi-lattice.

Each set which is not a superset of one of these two sets is pruned in line 6 of Algorithm 1.

In addition, we do not allow the creation of new constraints, this lead to consider left-hand sides of the tgds (2) and (4) as the upper-bound of our exploration space.

The explorable part of the resulting quasi-lattice is shown in Figure 3.

3.3.3 Exploring the quasi-lattice. During the exploration of the space of possible candidates, the user is challenged upon one element of the quasi-lattice at a time, as in line 10 of Algorithm 1.

This element can be chosen according to a given exploration strategy, corresponding to the call of SelectAtomSet in line 9. We will experimentally compare four different exploration strategies in Section 6.

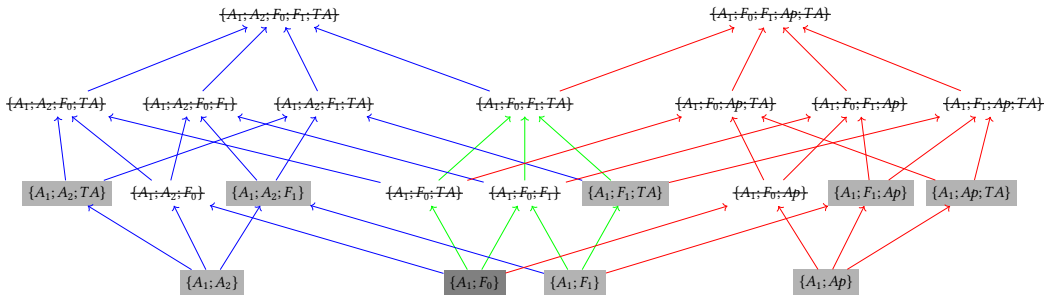


Fig. 3. Atom sets quasi-lattice on examples 3.9 and 3.10. With atoms: $A_1 = A(\text{idAir}, \text{name}_1, \text{town}_1)$, $A_2 = A(\text{idAir}', \text{name}_2, \text{town}_2)$, $F_0 = F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir})$, $F_1 = F(\text{idF}_1, \text{town}_1, \text{town}_2, \text{idAir}')$, $TA = TA(\text{idAg}, \text{name}_3, \text{town}_1)$ and $Ap = Airp(\text{idAp}, \text{name}_2, \text{town}_2)$.

An important property of the upper semilattice of atom refinement implies that, once the user validates one of the candidates, then all the supersets of such candidate can be excluded from further exploration, thus effectively pruning the search space.

Example 3.10. Following previous Example 3.9, we are refining tgds (2) and (4). Figure 3 illustrate the exploration space with the left side corresponding to atom sets specifics to tgd (2), the right side corresponding to atom sets specifics to tgd (4) and the central part corresponding to common atom sets between (2) and (4).

Assume, for the sake of the example, that we employ a breadth-first bottom-up strategy, starting the exploration of the upper semilattice in Figure 3 at its bottom-up level with $\{A_1; A_2\}$,$\{A_1; F_0\}$,$\{A_1; F_1\}$ and $\{A_1; Ap\}$. The user is asked about the validity of $\{A_1; A_2\}$ (the bottom left light gray box of Figure 3) with the following question:

> *"Are the tuples* $A(\texttt{a0}, \texttt{AA}, \texttt{L.A.})$ *and* $A(\texttt{a1}, \texttt{MAI}, \texttt{Miami})$ *enough to produce* $Dpt(\texttt{Miami}, \texttt{f2}, \texttt{c0})$, $Arr(\texttt{L.A.}, \texttt{f2}, \texttt{c0})$ *and* $Co(\texttt{c0}, \texttt{AA}, \texttt{L.A.})$*?"*

We can observe that a positive answer implies an ambiguity, namely that *the second flight company is based in the same town of the departure of the flight*, which is not the case in real-world examples. Hence, the user will be likely to answer 'No' to the above question.

Next, assume now that Algorithm 1 proceeds with $\{A_1; F_0\}$. This atom set is common between the tgds (2) and (4), consequently we can use tuples from (E_S^1, E_T^1) or (E_S^2, E_T^2) to generate the question. Here we take tuples from (E_S^1, E_T^1), leading to the following question:

> *"Are the tuples* $F(\texttt{f0}, \texttt{Miami}, \texttt{L.A.}, \texttt{a0})$ *and* $A(\texttt{a0}, \texttt{AA}, \texttt{L.A.})$ *enough to produce* $Dpt(\texttt{Miami}, \texttt{f2}, \texttt{c0})$, $Arr(\texttt{L.A.}, \texttt{f2}, \texttt{c0})$ *and* $Co(\texttt{c0}, \texttt{AA}, \texttt{L.A.})$*?"*

Assuming that the user will answer 'Yes' to this question, the supersets of $\{A_1; F_0\}$ will be pruned (crossed out boxes of Figure 3) and the following tgd will be output by the algorithm:

$$F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1) \\ \rightarrow \exists \text{idC}_0, \text{idF}_2, Dpt(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge Arr(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge Co(\text{idC}_0, \text{name}_1, \text{town}_1) \quad (5)$$

We continue the exploration of the current level with sets $\{A_1; F_1\}$ and $\{A_1; Ap\}$. Assuming that the user does not validate these sets, he will be finally challenged about the last available sets then on the next level of the semilattice, namely on the sets $\{A_1; A_2; TA\}$,$\{A_1; A_2; F_1\}$, $\{A_1; F_1; TA\}$, $\{A_1; F_1; Ap\}$and $\{A_1; Ap; TA\}$ which are also labels as invalid. In the end, for the combination of tgds (2) and (4), Algorithm 1 will output the single tgd (5).

We now state that when shifting from the initial canonical mapping to its refined form as given by Algorithm 1, we obtain a *more general* set of tgds.

Lemma 3.11. *Let $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ be a canonical mapping and let Σ' be a mapping obtained from atom refinement of $\mathcal{M}$, then, for all source instances E_S, there exists a morphism μ such that $\mu(\textsc{chase}(\Sigma, E_S)) \subseteq \textsc{chase}(\Sigma', E_S)$. By the correctness of the chase procedure, the logical entailment $\Sigma' \models \Sigma$ holds.*

Proof. For each tgd $\sigma = \phi \rightarrow \psi \in \mathcal{M}$ there exists at least one tgd $\sigma' = \phi' \rightarrow \psi \in \mathcal{M}'$ that is an atom refinement of σ. Then, ϕ' must correspond to a node in the semilattice, such that $\phi' \subseteq \phi$. We introduce an function $ref : \mathcal{M} \rightarrow \mathcal{M}'$ that associate to each σ in $\mathcal{M}$ one of its refinements (that may be arbitrarily chosen if there are several such tgds in $\mathcal{M}'$).

Let ν be an instantiation mapping to compute $\textsc{chase}(\mathcal{M}, E_S)$. That is, there exists a tgd $\sigma = \phi \rightarrow \psi \in \mathcal{M}$ such that $\nu(\phi) \subseteq E_S$ and $\nu(\psi) \subseteq \textsc{chase}(\mathcal{M}, E_S)$. Moreover each existential variable in ψ is mapped by ν to a fresh labeled null, which means that is ν^{-1} is defined for such values. Since $\phi' \subseteq \phi$, $\nu(\phi') \subseteq E_S$. Therefore, there exists an instantiation mapping ν' such that (1) $\nu'(\phi') \subseteq E_S$ (2) $\nu'(\psi') \subseteq \textsc{chase}(\mathcal{M}', E_S)$ and (3) for all variables x in ϕ', $\nu'(\mathrm{x}) = \nu(\mathrm{x})$. However, ν' and ν can differ in two ways: the domain of ν' can be smaller than the domain of ν and the labeled nulls that are assigned to existential variables in ψ can be different because the chase generate fresh null values at each tgd application. By construction of $\mathcal{E}_p$ in Algorithm 1, any variable x in ψ is either an existential variable or a universal variable in ϕ'. Thus, every variable x in ψ is either mapped to fresh null values by ν and ν' or, alternatively, $\nu(\mathrm{x}) = \nu'(\mathrm{x})$. We introduce μ_ν a morphism from $\nu(\psi) \subseteq \textsc{chase}(\mathcal{M}, E_S)$ to $\nu'(\psi) \subseteq \textsc{chase}(\mathcal{M}', E_S)$, defined as $\mu_\nu(c) = c$ if there exists x in ϕ' such that $\nu(\mathrm{x}) = c$ and $\mu_\nu(c) = \nu'(\nu^{-1}(c))$ otherwise (that if c is a fresh value generated by $\textsc{chase}(\mathcal{M}, E_S)$).

Let us consider two instantiation mappings ν_1 and ν_2 used in $\textsc{chase}(\mathcal{M}, E_S)$ and their associated morphisms μ_{ν_1} and μ_{ν_2}. Let c be a value in $dom(\mu_{\nu_1}) \cap dom(\mu_{\nu_2})$. If c is fresh and in $dom(\mu_{\nu_1})$, it means than it is the image of an existential variable by ν_1, which means that it cannot be the image of any variable by ν_2, and thus $c \notin dom(\mu_{\nu_2})$ which contradicts $c \in dom(\mu_{\nu_1}) \cap dom(\mu_{\nu_2})$. Thus c is not fresh, thus $\mu_{\nu_1}(c) = c = \mu_{\nu_2}(c)$. We define $\mu_{\{\nu_1,\nu_2\}}$ as $\mu_{\{\nu_1,\nu_2\}}(c) = \mu_{\nu_1}(c)$ if $c \in dom(\mu_{\nu_1})$ and $\mu_{\{\nu_1,\nu_2\}}(c) = \mu_{\nu_2}(c)$ otherwise. One can remark that $\mu_{\{\nu_1,\nu_2\}} \mid_{dom(\mu_{\nu_1})} = \mu_{\nu_1}$ and $\mu_{\{\nu_1,\nu_2\}} \mid_{dom(\mu_{\nu_2})} = \mu_{\nu_2}$. By iterating this construction on the finite set Λ of all instantiation mappings ν used in $\textsc{chase}(\mathcal{M}, E_S)$, we can build a morphism $\mu = \mu_\Lambda$.

Let t be a tuple in $\textsc{chase}(\mathcal{M}, E_S)$. There exists an instantiation morphism ν used in $\textsc{chase}(\mathcal{M}, E_S)$ and a tgd $\phi \rightarrow \psi$ such that $t \in \nu(\psi)$. Since $\mu_\nu(\nu(\psi)) \subseteq \textsc{chase}(\mathcal{M}', E_S)$ and $\mu \mid_{dom(\mu_\nu)} = \mu_\nu$ we deduce $\mu(t) \in \textsc{chase}(\mathcal{M}', E_S)$. □

The following Example 3.12 shows that the previous lemma would not hold if Algorithm 1 is allowed to create new existential variables.

Example 3.12. Given a pair (E_S, E_T) such that $E_S = \{R(\mathrm{x}, \mathrm{y}); S(\mathrm{z})\}$ and $E_T = \{T(\mathrm{x})\}$. The canonical mapping corresponding to (E_S, E_T) is $\Sigma = \{R(\mathrm{x}, \mathrm{y}) \wedge S(\mathrm{z}) \rightarrow T(\mathrm{x})\}$. Suppose that atom refinement allows the creation of existentially quantified variables. By applying this refinement on Σ, we may obtain the mapping $\Sigma' = \{S(\mathrm{z}) \rightarrow \exists x, T(\mathrm{x})\}$. Chasing E_S under Σ and Σ' will lead to following results:

$$\textsc{chase}(E_S, \Sigma) = \{T(\mathrm{x})\} \qquad \textsc{chase}(E_S, \Sigma') = \{T(\mathrm{x1})\}$$

for which there is no morphism μ such that $\mu(\textsc{chase}(E_S, \Sigma)) \subseteq \textsc{chase}(E_S, \Sigma')$, because the constant x has to be preserved.

The following Lemma 3.13 states that the intermediate mappings obtained after the atom refinement step is *split-reduced* and, at the opposite of the work in [13], have no *σ-redundant* tgds.

Lemma 3.13. *Given a normalized canonical mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$, application of atom refinement on the tgds in Σ always produces a mapping which is split-reduced and without σ-redundancy.*

Proof. As $\mathcal{M}$ is already normalized, it is *split-reduced*. During the refinement step, only atoms in the left-hand side are suppressed, so there is no way to break joins between existentially quantified variables as they are located only in the right-hand side. This means that $\mathcal{M}'$ is *split-reduced*.

Also, the refinement use one quasi-lattices for each block of ψ-equivalent tgds. So the only way to create equivalent tgds is to validate two equivalent left-hand sides conjunctions in a same quasi-lattice, and there is no equivalent nodes in such quasi-lattice. This mean that $\mathcal{M}'$ has no *σ-redundant* tgds.

□

3.3.4 Questioning about atoms set validity. In the atom refinement algorithm, the user is challenged on the validity of the left-hand side atoms of the canonical mapping at line 10 of Algorithm 1.

We build on the correspondence between these atoms and the tuples that appear in the sources ${E_S}^i$ to ask pertinent questions, as those shown in Example 3.10. The AskAtomSetValidity(e, ψ_b) subroutine that appears in Algorithm 1 constructs a pair $({E_S}^{e,\psi_b}, {E_T}^{e,\psi_b})$ by transforming the candidate subset e into ${E_S}^{e,\psi_b}$, formally ${E_S}^{e,\psi_b} = \{\bar{\theta}(a) | a \in e\}$). Then the chase procedure is used to compute ${E_T}^{e,\psi_b}$, formally ${E_T}^{e,\psi_b} = \text{chase}(e \rightarrow \psi_b, {E_S}^{e,\psi_b})$.

Example 3.14. This example focuses on the generation of the exemplar tuples underlying the questions of Example 3.10 while refining the tgds (2) and (4). We are challenging the user about the validity of the set of atoms $e = \{F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0); A(\text{idAir}_0, \text{name}_1, \text{town}_1)\}$, which is a subset of the left-hand side of the tgds (2) and (4). For each tgd, these atoms are built from the sets $E_S^{1'} = \{F(\texttt{f0}, \texttt{Miami}, \texttt{L.A.}, \texttt{a0}); A(\texttt{a0}, \texttt{AA}, \texttt{L.A.})\}$ and $E_S^{2'} = \{F(\texttt{f0}, \texttt{Lyon}, \texttt{Paris}, \texttt{a0}); A(\texttt{a0}, \texttt{AF}, \texttt{Paris})\}$,respectively a subset the of instances E_S^1 and E_S^2. We want to challenge the user whether the following generalization of the tgds (2) and (4) is sufficient:

$$\sigma = F(\text{idF}_0, \text{town}_2, \text{town}_1, \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1)$$
$$\rightarrow \exists \text{idC}_0, \text{idF}_2, \mathit{Dpt}(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge \mathit{Arr}(\text{town}_1, \text{idF}_2, \text{idC}_0) \wedge \mathit{Co}(\text{idC}_0, \text{name}_1, \text{town}_1)$$

The chase procedure applies σ on $E_S^{1'}$ (resp. $E_S^{2'}$) to obtain the following instance $E_T^{1'}$ (resp. $E_T^{2'}$), from which the first question appearing in Example 3.10 is derived:

$$E_T^{1'} = \{\mathit{Dpt}(\texttt{Miami}, \texttt{f2}, \texttt{c0}); \mathit{Arr}(\texttt{L.A.}, \texttt{f2}, \texttt{c0}); \mathit{Co}(\texttt{c0}, \texttt{AA}, \texttt{L.A.})\}$$
$$E_T^{2'} = \{\mathit{Dpt}(\texttt{Lyon}, \texttt{f2}, \texttt{c0}); \mathit{Arr}(\texttt{Paris}, \texttt{f2}, \texttt{c0}); \mathit{Co}(\texttt{c0}, \texttt{AF}, \texttt{Paris})\}$$

3.4 Join refinement between variables of a tgd

In relational data, multiple occurrences of the same value do not necessarily imply a semantic relationship between the attributes containing such a value. An example from our running scenario is the occurrence of the constant L.A. both as the city where an airline company is located, and as the arrival and departure city of flights booked by that airline company. However, the canonical mapping imposes such co-occurrences that may be due to spurious use of the same variable. Thus, the canonical mapping may introduce irrelevant joins in the left-hand side of the tgds. In order to produce the mapping the user has in his mind, we primarily need to distinguish relevant joins from irrelevant ones. This section presets the join refinement step and details the join Algorithm 2 that explores the candidate joins in each tgd by inquiring the user about the validity of such joins.

As joins in conjunctive queries are encoded by multiple occurrences of a variable, we refer to these variables as to *join variables*, refining a join corresponds to replace some occurrences with

Algorithm 2 TgdsJoinRefinement(Σ)

Input: A set of tgds Σ to be join refined.
Output: A set of tgds Σ' where each tgd is join refined.

```
1: Σ' ← ∅
2: for all σ ∈ Σ do
3:     let σ = φ(x̄) → ∃ȳ, ψ(x̄, ȳ)
4:     Σ_t ← {σ}
5:     for all x ∈ x̄ do
6:         if variable x occurs more than once in φ then
7:             Σ_explored ← Σ_t
8:             Σ_t ← ∅
9:             for all σ' ∈ Σ_explored do
10:                Σ_t ← Σ_t ∪ VarJoinsRefinement(Σ_t, σ', x)
11:            end for
12:        end if
13:    end for
14:    Σ' ← Σ' ∪ Σ_t
15: end for
16: return Σ'
```

fresh variables. In Algorithm 2 this replacement of join variables by fresh ones is conducted by the subroutine named VarJoinsRefinement which is detailed in Algorithm 3. The subroutine explores the partitions of these newly introduced variables and questions the user to check if the joins are relevant (some fresh variables are unified) or not (they are kept renamed). A block in the set of all partitions represents the variables to be unified together.

Example 3.15. Recall tgd (5) from Example 3.10 obtained after the atom-refined mapping below:

$$F(\mathrm{idF}_0, \mathrm{town}_2, \mathrm{town}_1, \mathrm{idAir}_0) \wedge A(\mathrm{idAir}_0, \mathrm{name}_1, \mathrm{town}_1) \\ \rightarrow \exists \mathrm{idC}_0, \mathrm{idF}_2, \mathit{Dpt}(\mathrm{town}_2, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Arr}(\mathrm{town}_1, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Co}(\mathrm{idC}_0, \mathrm{name}_1, \mathrm{town}_1) \quad (5)$$

There is an ambiguity on the use of the same town as the town of arrival and departure of flights and the town where a travel agency is located, as shown by the multiple occurrences of the join variable town_1 at four different positions. Each occurrence of town_1 is replaced with a fresh variable (namely town_1', town_1'', town_1''' and town_1'''') yielding the following candidate tgd:

$$F(\mathrm{idF}_0, \mathrm{town}_2, \mathrm{town}_1', \mathrm{idAir}_0) \wedge A(\mathrm{idAir}_0, \mathrm{name}_1, \mathrm{town}_1'') \\ \rightarrow \exists \mathrm{idC}_0, \mathrm{idF}_2, \mathit{Dpt}(\mathrm{town}_2, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Arr}(\mathrm{town}_1''', \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Co}(\mathrm{idC}_0, \mathrm{name}_1, \mathrm{town}_1'''')$$

In the corresponding quasi-lattice of the set $\{\mathrm{town}_1', \mathrm{town}_1'', \mathrm{town}_1''', \mathrm{town}_1''''\}$, the upper-bound corresponds to the case where no refinement is needed, all occurrences being replaced with the original town_1.

Given a variable x in a tgd $\sigma = \phi \rightarrow \psi$, we consider the set of its occurrences in $\phi \cup \psi$. Since we do not wish to introduce new existentially quantified variables, each variable occurrence in ψ must be bound to at least one variable occurrence in ϕ. In order to achieve this, we only consider the partitions in which all blocks contain at least one occurrence in ϕ. Those partitions are called *well-formed*.

Well-formed partitions are equipped with a quasi-lattice structure: given two partitions $\mathcal{P}$ and $\mathcal{P}'$, if $\mathcal{P} \leq \mathcal{P}'$ and $\mathcal{P}$ is well-formed, then $\mathcal{P}'$ is well-formed as well. In particular, if $\mathcal{P} \leq \mathcal{P}'$ then

all unifications encoded by $\mathcal{P}$ are also performed encoded in $\mathcal{P}'$. This means that if $\mathcal{P}$ is acceptable for the user, then it is also the case for $\mathcal{P}'$. Conversely, if $\mathcal{P}'$ is not acceptable for the user (i.e., some joins are missing), then neither is $\mathcal{P}$. We employ these criteria to prune the search space during the exploration of the quasi-lattice of occurrences of x. This quasi-lattice structure allow us to avoid exploration of partitions leading to redundant tgd, which is not the case with the use of separate semilattices as used in [13].

Example 3.16. Following Example 3.15, town_1''' and town_1'''' must be in a partition containing either town_1' or town_1''. This means that partitions containing one of the blocks $\{\text{town}_1'''\}$, $\{\text{town}_1''''\}$ or $\{\text{town}_1''', \text{town}_1''''\}$ are not well-formed and will be excluded.

Algorithm 3 *Subroutine:*VarJoinsRefinement(Σ_t, σ, x)

Input: A set of previously join refined tgds Σ_t.
Input: A tgd σ.
Input: A variable x $\in \sigma$ on which the refinement is made.
Output: A set of tgds Σ_{out} of join refinements of σ for variable x.

1: generate from σ a tgd σ' where occurrences of x are renamed with fresh variables and a morphism μ_{orig} such that $\mu_{orig}(\sigma') = \sigma$
2: **let** $\sigma' = \phi' \rightarrow \psi'$
3: $\mathcal{J}_{cand} \leftarrow$ generate set of possibles candidates join partitions from σ'
4: $\mathcal{J}_v \leftarrow$ generate supremum of the join lattice from σ'
5: **while** $\mathcal{J}_{cand} \neq \emptyset$ **do**
6: $\quad \mathcal{P} \leftarrow$ SelectPartition($\mathcal{J}_{cand}, \mathcal{J}_v$)
7: $\quad \sigma'' \leftarrow$ UnifyVariables($\sigma', \mathcal{P}$)
8: $\quad$ **if** $(\nexists \sigma_t \in \Sigma_t, \sigma_t \models \sigma'') \wedge$ AskJoinsValidity(σ'') **then**
9: $\quad\quad$ add $\mathcal{P}$ to $\mathcal{J}_v$
10: $\quad\quad$ remove upper partitions of $\mathcal{P}$ from $\mathcal{J}_v$
11: $\quad\quad$ remove $\mathcal{P}$ and its upper partitions from $\mathcal{J}_{cand}$
12: $\quad$ **else**
13: $\quad\quad$ remove $\mathcal{P}$ and its lower partitions from $\mathcal{J}_{cand}$
14: $\quad$ **end if**
15: **end while**
16: $\Sigma_{out} \leftarrow \emptyset$
17: **for all** $\mathcal{P} \in \mathcal{J}_v$ **do**
18: $\quad \sigma'' \leftarrow$ UnifyVariables($\sigma', \mathcal{P}$)
19: $\quad$ add σ'' to Σ_{out}
20: **end for**
21: **return** Σ_{out}

Algorithm 2 implements the join refinement by iterating variable refinements on each universal variable of each tgd. As we do not consider the possibility of creating new joins, but only the suppression of joins which already exists, each original variable is considered separately. However, since each call to VarJoinsRefinement may generate multiple refined tgds, one for each refined join variable, we need to combine these refinements. This is done by verifying that the join partition currently evaluated does not lead to produce a tgd which is redundant or prunable. This is done in Algorithm 3 at line 8, by checking if there's another tgd $\sigma_t \in \Sigma_t$ which is a model of the currently evaluated tgd (i.e. σ_t is logically equivalent or more general than the evaluated tgd).

Subroutine VarJoinsRefinement(σ, x) explores the part of the quasi-lattice of a variable x corresponding to a tgd σ, asking questions to the user in order to determine the proper join refinement. In line 1, occurrences of x are replaced with fresh variables yielding a tgd σ' and a morphism μ_{orig} such that $\mu_{orig}(\sigma') = \sigma$. Line 3 initializes the quasi-lattice by excluding malformed partitions as stated above. The SelectPartition subroutine selects a partition in the set of partitions and encodes the specific exploration strategy on top of the quasi-lattice. Any suitable exploration strategy can be plugged in here, as shown in the experimental study presented in Section 6. Function UnifyVariables($\sigma, \mathcal{P}$) (lines 7 and 18 of the Algorithm) returns a tgd corresponding to σ where variables from the same block of a partition $\mathcal{P}$ are unified. The user is asked about the validity of this unification in line 8 and the search space and results are pruned according to his answer in lines 10, 11 and 13. One can easily prove the following Lemma, which is the counterpart of Lemma 3.11 for join refinement. Hence, Lemma 3.17 establishes the logical entailment of the join-refined mapping.

Lemma 3.17. *Let Σ be a mapping and let Σ' be a mapping obtained from Σ after join refinement, then $\Sigma' \models \Sigma$.*

Proof. Let $\sigma = \phi \rightarrow \psi$ be a tgd and x be a universal variable in σ. First, we prove that for all $\sigma'' \in$ VarJoinsRefinement(σ, x), $\sigma'' \models \sigma$.

Let $\sigma' = \phi' \rightarrow \psi'$ be the tgd obtained from σ by replacing occurrences of x with a fresh variable, and μ_{orig} be the morphism such that $\mu_{orig}(\sigma') = \sigma$. Let $\sigma'' = \phi' \rightarrow \psi'$. As σ'' results from the unification of fresh variables in σ', there is a morphism μ_{unif} such that $\mu_{unif}(\sigma') = \sigma''$. Let $\mu_{\sigma''}$ be the morphism defined by: $\mu_{\sigma''}(\mathrm{y}) = \mathrm{x}$ if y results from the unification of fresh variables in σ', $\mu_{\sigma''}(\mathrm{y}) = \mathrm{y}$ otherwise. By construction, $\mu_{\sigma''}(\sigma'') = \sigma$. One can remark that existential variables in ψ'' are the same as the ones in ψ, thus $\mu_{\sigma''}$ is injective for these variables.

In Algorithm 2, Σ_t contains tgds that are either elements of Σ or obtained by applying VarRefinement to previous elements of Σ_t. Because of line 8, VarRefinement always returns at least one tgd. Thus, for each initial tgd σ in Σ, there is a tgd σ' in Σ' coming from successive calls of VarRefinement starting with σ. By transitivity of $\models$ we deduce that $\sigma' \models \sigma$. Thus, $\Sigma' \models \sigma$. Since this holds for all tgds in Σ, we conclude that $\Sigma' \models \Sigma$. □

Example 3.18. We recall the tgd (5) from Example 3.10:

$$F(\mathrm{idF}_0, \mathrm{town}_2, \mathrm{town}_1, \mathrm{idAir}_0) \wedge A(\mathrm{idAir}_0, \mathrm{name}_1, \mathrm{town}_1) \\ \rightarrow \exists \mathrm{idC}_0, \mathrm{idF}_2, \mathit{Dpt}(\mathrm{town}_2, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Arr}(\mathrm{town}_1, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Co}(\mathrm{idC}_0, \mathrm{name}_1, \mathrm{town}_1) \tag{5}$$

Its set of universal variables is $\overline{x} = \{\mathrm{idF}_0, \mathrm{town}_2, \mathrm{town}_1, \mathrm{idAir}_0, \mathrm{name}_1\}$. As Algorithm 2 only considers variables that appears several times (line 6), we only consider town_1 and idAir_0 of $\overline{x}$. Considering first the idAir_0 variable, a renaming of each of its occurrences to $\mathrm{idAir}_0{}'$ and $\mathrm{idAir}_0{}''$ leads to the following tgd:

$$F(\mathrm{idF}_0, \mathrm{town}_2, \mathrm{town}_1, \mathrm{idAir}_0{}') \wedge A(\mathrm{idAir}_0{}'', \mathrm{name}_1, \mathrm{town}_1) \\ \rightarrow \exists \mathrm{idC}_0, \mathrm{idF}_2, \mathit{Dpt}(\mathrm{town}_2, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Arr}(\mathrm{town}_1, \mathrm{idF}_2, \mathrm{idC}_0) \wedge \mathit{Co}(\mathrm{idC}_0, \mathrm{name}_1, \mathrm{town}_1) \tag{6}$$

The quasi-lattice contains two partitions $\{\{\mathrm{idAir}_0{}'\}; \{\mathrm{idAir}_0{}''\}\}$ and $\{\{\mathrm{idAir}_0{}'; \mathrm{idAir}_0{}''\}\}$. The user is asked about the validity of $\{\{\mathrm{idAir}_0{}'\}; \{\mathrm{idAir}_0{}''\}\}$, *i.e.*, to have the identifier of an airline company unrelated to its flight. The user will likely answer 'No' to the above question, thus keeping the upper-bound $\{\{\mathrm{idAir}_0{}'; \mathrm{idAir}_0{}''\}\}$ of the quasi-lattice valid. Since these join is relevant, the tgd is not modified.

Then, we consider the town_1 variable. A renaming of each of its occurrences leads to the following tgd previously given in Example 3.15:

$$F(\text{idF}_0, \text{town}_2, \text{town}_1', \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1'') \\ \rightarrow \exists \text{idC}_0, \text{idF}_2, \mathit{Dpt}(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge \mathit{Arr}(\text{town}_1''', \text{idF}_2, \text{idC}_0) \wedge \mathit{Co}(\text{idC}_0, \text{name}_1, \text{town}_1'''')$$

There are five partitions that do not create new existential variables, namely :

$$\{\{\text{town}_1'; \text{town}_1'''\}; \{\text{town}_1''; \text{town}_1''''\}\}, \{\{\text{town}_1'; \text{town}_1''''\}; \{\text{town}_1''; \text{town}_1'''\}\}, \\ \{\{\text{town}_1'; \text{town}_1'''; \text{town}_1''''\}; \{\text{town}_1''\}\}, \{\{\text{town}_1'\}; \{\text{town}_1''; \text{town}_1'''; \text{town}_1''''\}\} \\ \text{and } \{\{\text{town}_1'; \text{town}_1''; \text{town}_1'''; \text{town}_1''''\}\}.$$

The user is asked about the validity of the candidate partition $\{\{\text{town}_1'; \text{town}_1'''\}; \{\text{town}_1''; \text{town}_1''''\}\}$ with the following question:

"Are the tuples $F(\texttt{f0}, \texttt{Miami}, \texttt{L.A.}', \texttt{a0})$ *and* $A(\texttt{a0}, \texttt{AA}, \texttt{L.A.}'')$ *enough to produce* $\mathit{Dpt}(\texttt{Miami}, \texttt{f2}, \texttt{c0})$, $\mathit{Arr}(\texttt{L.A.}', \texttt{f2}, \texttt{c0})$ *and* $\mathit{Co}(\texttt{c0}, \texttt{AA}, \texttt{L.A.}'')$*?"*

Since this partition is acceptable for the user, he will probably answer 'Yes'. Therefore, the upper-bound $\{\{\text{town}_1'; \text{town}_1''; \text{town}_1'''; \text{town}_1''''\}\}$ of the quasi-lattice is pruned and the following tgd is added to the output:

$$F(\text{idF}_0, \text{town}_2, \text{town}_1', \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1'') \\ \rightarrow \exists \text{idC}_0, \text{idF}_2, \mathit{Dpt}(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge \mathit{Arr}(\text{town}_1', \text{idF}_2, \text{idC}_0) \wedge \mathit{Co}(\text{idC}_0, \text{name}_1, \text{town}_1'')$$

The exploration continues with the remaining candidate partitions. However, as the remaining partitions either relate an airline's headquarters to an arrival or a flight to a company's headquarters, the user will consistently answer 'No' to these questions.

As formalized in the following Lemma 3.19, the join refinement step preserves the *split-reduction* property of mappings and, at the opposite of the work in [13], does not might introduce σ*-redundant* tgds. Hence, similarly to the atom refinement step and its associated Lemma 3.13, a normalization step following join refinement is not necessary.

Lemma 3.19. *Given a normalized mapping* $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$*, application of join refinement on the tgds in* Σ *always produces a mapping which is normalized.*

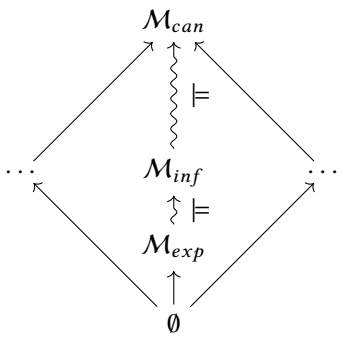
Proof. By definition, if a tgd σ is *split-reduced* and contain more than one atom in its right-hand side, these atoms (at least two) are joined using existentially quantified variables. Since join refinement only focuses on universal variables, existential variables are preserved. Thus, all atoms in the right-hand side of join refined tgds are joined together using these existential variables, which means that join refined tgds are also *split-reduced*.

As Σ is normalized, each of its tgd is *split-reduced*. Since for each tgd in Σ, the application of the join refinement step results in new tgds that are also *split-reduced*. Thus, the set Σ' of all these refined tgds is a *split-reduced* mapping.

As each tgd is produced only if there's no logically equivalent tgd previously produced (Algorithm 3 at line 8), then no additional step of σ*-redundancy suppression* is needed.

As the mapping produced is *split-reduced* and does not contain σ*-redundancy*, then it is normalized. □

In the join refinement step, the suppression of joins can generate additional tuples in the target instance. For such a reason, similarly to the generation of questions in the atom refinement step,



Given a mapping $\mathcal{M}_{exp}$ expected by an user :

$\mathcal{M}_{inf} \models \mathcal{M}_{can}$ is proved in Theorem 4.5

$\mathcal{M}_{exp} \models \mathcal{M}_{inf}$ is proved in Theorem 4.6

Confluence to a mapping $\mathcal{M}_{final}$ is proved in Theorem 4.7

Convergence is proved in Theorem 4.8

If a *fully informative* exemplar tuples set is provided :

$\mathcal{M}_{final} \equiv \mathcal{M}_{exp}$ is proved in Theorem 4.14

Fig. 4. Summary of the main theorems about our framework (in Section 4).

the source instance is again chased to generate such additional tuples [2]. Similarly to the subroutine described in Section 3.3.4 for atom refinement, the AskJoinsValidity subroutine that appears in Algorithm 2 constructs a pair $(E_S{}^{\sigma}, E_T{}^{\sigma})$ by instantiating the left-hand side of a candidate tgd σ to obtain a source instance $E_S{}^{\sigma}$ and then chasing it to build $E_T{}^{\sigma}$.

Example 3.20. We illustrate the questions asked to the user in Example 3.18. We challenge the user on the validity of the partition $\mathcal{P} = \{\{\text{town}_1\text{'}; \text{town}_1\text{'''}\}; \{\text{town}_1\text{''}; \text{town}_1\text{''''}\}\}$ in the following tgd:

$$\sigma = F(\text{idF}_0, \text{town}_2, \text{town}_1\text{'}, \text{idAir}_0) \wedge A(\text{idAir}_0, \text{name}_1, \text{town}_1\text{''})$$
$$\rightarrow \exists \text{idC}_0, \text{idF}_2, \mathit{Dpt}(\text{town}_2, \text{idF}_2, \text{idC}_0) \wedge \mathit{Arr}(\text{town}_1\text{'}, \text{idF}_2, \text{idC}_0) \wedge \mathit{Co}(\text{idC}_0, \text{name}_1, \text{town}_1\text{''})$$

The instance $E_S{}^{\sigma}$ obtained from the left-hand side of σ through the bijection $\bar{\theta}$ is the following:

$$E_S{}^{\sigma} = \{F(\texttt{f0}, \texttt{Miami}, \texttt{L.A.}', \texttt{a0}); A(\texttt{a0}, \texttt{AA}, \texttt{L.A.}'')\}$$

Chasing $E_S{}^{\sigma}$ with σ leads to:

$$E_T{}^{\sigma} = \{\mathit{Dpt}(\texttt{Miami}, \texttt{f2}, \texttt{c0}); \mathit{Arr}(\texttt{L.A.}', \texttt{f2}, \texttt{c0}); \mathit{Co}(\texttt{c0}, \texttt{AA}, \texttt{L.A.}'')\}$$

Those exemplar tuples are finally rewritten into questions as shown in Example 3.18.

At the end of this step, the mapping $\mathcal{M}_{final}$ is returned to the user as the result of the framework execution.

4 FORMAL GUARANTEES OF THE FRAMEWORK

In this section, we prove the correctness and the completeness of our interactive mapping specification framework.

First, we describe the set of questions that can be asked by our framework and the transition rule that describes how the framework rewrites the mapping at each iteration. Then, we show that if the user provides a set of exemplar tuples for his expected mapping, then our framework will converge to a single mapping in the space of all possible inferred mappings. Finally, we show that if the user provides a set of exemplar tuples that *fully describe* the mapping that he has in mind, then our framework will always return a logically equivalent mapping to the latter mapping. The

[2]We recall that the chase is polynomial for Σ consisting of only s-t tgds. Thus, repeating it several times as additional tuples come, is appropriate.

mains theorems of this section are summarized in Figure 4, which offers a guideline for the reader through the main theorems.

The set of all possible tgds explored by our process, and the set of questions about the validity of those tgds are expressed as follows :

Definition 4.1 (Explored set of candidates tgds). Let $\mathcal{M}_{can}$ be a canonical mapping. The set of candidates tgds is defined as follows :

$$\mathcal{M}_{candidates} = \bigcup_{(\phi \rightarrow \psi) \in \mathcal{M}_{can}} \{\phi' \rightarrow \psi' | \phi' \neq \emptyset \wedge \psi' \neq \emptyset \wedge (\exists \mu \text{ such that } \mu(\phi') \subseteq \phi \wedge \mu(\psi') \subseteq \psi)\}$$

Definition 4.2 (Set of asked questions). Let $\mathcal{M}_{can}$ be a canonical mapping. Let $\mathcal{M}_{candidates}$ be the set of tgds explored by our framework for $\mathcal{M}_{can}$. The set of questions $\mathcal{Q}$ that can be asked by our framework is the set :

$$\mathcal{Q} = \{\text{“Are the tuples } \bar{\theta}(\phi) \text{ enough to produce } \bar{\theta}(\psi)?\text{”} \mid (\phi \rightarrow \psi) \in \mathcal{M}_{candidates}\}$$

Given a previously inferred mapping, a question and the oracle's answer to this question, our framework will produce a new mapping. This is expressed by the following transition rule :

Definition 4.3 (Transition rule). Let $\mathcal{M}_{inf}$ be a mapping. Let q be a question about the validity of the tgd $\phi \rightarrow \psi$.

Then we have :

$$\mathcal{M} \xrightarrow{q} \mathcal{M}' \text{ such that : } \texttt{if } answer(q, Oracle) \texttt{ then } \mathcal{M}' = \mathcal{M} \cup (\phi \rightarrow \psi) \texttt{ else } \mathcal{M}' = \mathcal{M}$$

Remark 1. *The framework is non-deterministic as there are multiple possible questions q that can be asked from a single mapping $\mathcal{M}$.*

Our framework uses the canonical mapping, computed from the user's set of exemplar tuples, as the initial mapping. Then, our framework iteratively rewrites this mapping by asking the questions in the set $\mathcal{Q}$ defined in definition 4.2. This exploration can be expressed as a succession of applications of transition rule over $\mathcal{Q}$. More formally :

Definition 4.4 (Exploration). Let $\mathcal{M}_{can}$ be a canonical mapping. Let $\mathcal{Q}$ be the set of questions that can be asked over $\mathcal{M}_{can}$.

Then, an exploration of the set $\mathcal{Q}$ is a series of applications of the transition rule :

$$\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{inf} \text{ such that } \{q_1; \ldots; q_n\} \in \mathcal{Q}$$

$\mathcal{M}_{inf}$ is called an *inferred mapping*.

Correctness of the framework. We will now show that, given an expected mapping $\mathcal{M}_{exp}$ and a canonical mapping $\mathcal{M}_{can}$ obtained from a set of exemplar tuples for $\mathcal{M}_{exp}$, then every mapping $\mathcal{M}_{inf}$ inferred by our framework is such that $\mathcal{M}_{exp} \models \mathcal{M}_{inf} \models \mathcal{M}_{can}$.

First, we show that the inferred mappings imply the canonical mapping. This comes from the fact that our framework will relax constraints of the canonical mapping, without introducing new ones.

Theorem 4.5. *Let $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{inf}$ be an exploration. Then :*

$$\mathcal{M}_{inf} \models \mathcal{M}_{can}$$

Proof. This theorem follows from the definition 4.3. This definition shows that consecutive applications of the rewriting rules will only add new tgds without suppressing tgds in $\mathcal{M}_{can}$. Thus, $\mathcal{M}_{inf} \supseteq \mathcal{M}_{can}$, from which follow $\mathcal{M}_{inf} \models \mathcal{M}_{can}$. □

In addition to this theorem, we show that our framework will only produce mappings implied by the expected mapping. This guarantees that the inferred mapping will not produce extraneous tuples that would not be produced by the expected mapping.

Theorem 4.6. *Let $\mathcal{M}_{exp}$ be the expected mapping by Oracle. Let $\mathcal{E}$ be a set of exemplar tuples for $\mathcal{M}_{exp}$. Let $\mathcal{M}_{can}$ be the canonical mapping computed from $\mathcal{E}$. Let $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{inf}$ be an exploration. Then :*

$$\mathcal{M}_{exp} \models \mathcal{M}_{inf}$$

Proof. The proof is done by induction over an exploration :

- Suppose that we have an exploration $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{inf}$ such that $\mathcal{M}_{exp} \models \mathcal{M}_{inf}$. The application of a new rewriting rule will lead to the following exploration :

  $$\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{inf} \xrightarrow{q_{n+1}} \mathcal{M}'_{inf}$$

  with q_{n+1} being a question over a tgd $\phi \rightarrow \psi$.
  - if $answer(q_{n+1}, Oracle) = \texttt{false}$ then, by definition 4.3, $\mathcal{M}'_{inf} = \mathcal{M}_{inf}$. By the induction hypothesis : $\mathcal{M}_{exp} \models \mathcal{M}'_{inf}$.
  - if $answer(q_{n+1}, Oracle) = \texttt{true}$ then, by definition 4.3, $\mathcal{M}'_{inf} = \mathcal{M}_{inf} \cup \{\phi \rightarrow \psi\}$. Also, by definition 3.7, if the oracle answer `true` to q_{n+1} then $\psi \subseteq \textsc{chase}(\phi, \mathcal{M}_{exp})$, i.e. $\mathcal{M}_{exp} \models \{\phi \rightarrow \psi\}$. Since $\mathcal{M}_{exp} \models \mathcal{M}_{inf}$ by induction hypothesis, we obtain $\mathcal{M}_{exp} \models \mathcal{M}'_{inf}$.
- From $\mathcal{E}$ we have the non-normalized canonical mapping :

  $$\mathcal{M}_{can\_raw} = \{\bar{\theta}^{-1}(E_S) \rightarrow \bar{\theta}^{-1}(E_T) \mid (E_S, E_T) \in \mathcal{E}\}$$

  By definition 3.1 of the exemplar tuples we also have :

  $$\forall (E_S, E_T) \in \mathcal{E}, E_T \subseteq \textsc{chase}(\mathcal{M}_{exp}, E_S)$$

  As $\bar{\theta}^{-1}$ is an isomorphism, we can do the following substitution :

  $$\forall (\bar{\theta}^{-1}(E_S) \rightarrow \bar{\theta}^{-1}(E_T)) \in \mathcal{M}_{can\_raw}, \bar{\theta}^{-1}(E_T) \subseteq \textsc{chase}(\mathcal{M}_{exp}, \bar{\theta}^{-1}(E_S))$$

  which means, by definition of mapping implication, that $\mathcal{M}_{exp} \models \mathcal{M}_{can\_raw}$.
  By correctness of the normalization rules (*split-reduction* and *σ-redundancy suppression* [23]), we have $\mathcal{M}_{can} \equiv \mathcal{M}_{can\_raw}$, and thus $\mathcal{M}_{exp} \models \mathcal{M}_{can}$.

□

Moreover, our framework will always converge to one unique mapping regardless of the order in which questions are asked. This is shown in the following theorems showing the confluence and the convergence of our framework :

Theorem 4.7 (Confluence). *Let $\mathcal{M}$ be a mapping. Let $\mathcal{M} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_n$ and $\mathcal{M} \xrightarrow{q'_1} \ldots \xrightarrow{q'_m} \mathcal{M}_m$ be two explorations from $\mathcal{M}$.*
Then there exists a mapping $\mathcal{M}'$ such that we can find two explorations :

$$\mathcal{M}_n \xrightarrow{q_{n+1}} \ldots \xrightarrow{q_{n+k}} \mathcal{M}' \text{ and } \mathcal{M}_m \xrightarrow{q'_{m+1}} \ldots \xrightarrow{q'_{m+k'}} \mathcal{M}'$$

Proof. When a question is asked, the tgd corresponding to this question will be added or not to the inferred mapping, depending on the oracle answer. This process is completely independent from the previously asked questions and does not modify the set of questions that are asked. Therefore, the order in which questions are asked does not influence the result. Thus, it is easy to construct the two sets questions $\{q_{n+1}; \ldots; q_{n+k}\}$ and $\{q'_{m+1}; \ldots; q'_{m+k}\}$ as follows :

$$\{q_{n+1}; \ldots; q_{n+k}\} = \{q'_{m+1}; \ldots; q'_{m+k'}\} \setminus \{q_1; \ldots; q_n\}$$
$$\{q'_{m+1}; \ldots; q'_{m+k}\} = \{q_{n+1}; \ldots; q_{n+k}\} \setminus \{q'_1; \ldots; q'_m\}$$

□

Theorem 4.8 (Convergence). *Let* $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_k} \mathcal{M}_k \xrightarrow{q_{k+1}} \ldots$ *be an infinite exploration. Then :*

$$\exists k \in \mathbb{N} \text{ such that } \forall k' \geq k, \mathcal{M}_k \equiv \mathcal{M}_{k'}$$

Proof. This follows from definition 4.2 of the set of all questions that can be asked, and from Theorem 4.7. If the whole set of questions is explored, then asking one of this question one more time, or asking a question isomorphic to a question of this set, will only lead to an equivalent mapping. □

Following from the convergence theorem, we can define a *complete exploration* for our framework as follows :

Definition 4.9 (Complete exploration). Let $\mathcal{M}_{can}$ be a canonical mapping. Let $\mathcal{Q}$ be the set of questions that can be asked over $\mathcal{M}_{can}$.

Then, a complete exploration of the set $\mathcal{Q}$ is a series of applications of the transition rules :

$$\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{final} \text{ where } \{q_1; \ldots; q_n\} \in \mathcal{Q}$$

such that :

$$\forall q \in \mathcal{Q}, \mathcal{M}_{final} \xrightarrow{q} \mathcal{M}_{final}$$

Completeness of the framework. If an user produce a set of exemplar tuples that does not contain the necessary information to derive each tgd in his expected mapping, then it's easily seen that the final mapping obtained after a complete exploration cannot be equivalent to the expected one. However, we will now show that if the user provides a *fully informative* set of exemplar tuples for his expected mapping, our framework will always return a mapping logically equivalent to the user's expected mapping.

To do so, we will first show that for every expected mapping, there exists an ideal set of questions such that if they are all asked to the user, then the framework will return a mapping logically equivalent to the expected mapping. Then we show that, for every *fully informative* set of exemplar tuples for the user's expected mapping, our framework will always ask the ideal set of questions.

Definition 4.10 (Ideal exemplar tuples set). Let $\mathcal{M}$ be a mapping. Let $\mathcal{E}$ be a set of exemplar tuples for $\mathcal{M}$. Then $\mathcal{E}$ is an *ideal exemplar tuples set* if the canonical mapping $\mathcal{M}_{can}$ extracted from $\mathcal{E}$ is such that $\mathcal{M}_{can} \equiv \mathcal{M}$.

Lemma 4.11. *For all GLAV mapping* $\mathcal{M}$*, there exists an ideal exemplar tuples set* $\mathcal{E}_{ideal}$*.*

Proof. W.l.o.g., we suppose that $\mathcal{M}$ is normalized. From $\mathcal{M}$ we can construct a set :

$$\mathcal{E} = \{(\bar{\theta}(\phi), \bar{\theta}(\psi)) | (\phi \rightarrow \psi) \in \mathcal{M}\}$$

As each exemplar tuple $(E_S, E_T) \in \mathcal{E}$ come directly from a tgd $\sigma \in \mathcal{M}$, it follows that $E_T \subseteq \text{chase}(\sigma, E_S)$. It follows that $\mathcal{E}$ is an exemplar tuples set for $\mathcal{M}$.

Also, as the non-normalized canonical mapping $\mathcal{M}_{can\_raw}$ is obtained by applying the morphism from tuples to atoms $\bar{\theta}^{-1}$ to each exemplar tuple in $\mathcal{E}$ and by definition of $\mathcal{E}$, we obtain :

$$\begin{aligned}\mathcal{M}_{can\_raw} &= \{(\bar{\theta}^{-1}(\bar{\theta}(\phi)) \rightarrow \bar{\theta}^{-1}(\bar{\theta}(\psi)))|(\phi \rightarrow \psi) \in \mathcal{M}\} \\ &= \{(\phi \rightarrow \psi)|(\phi \rightarrow \psi) \in \mathcal{M}\} \\ &= \mathcal{M}\end{aligned}$$

We know than the normalization of $\mathcal{M}_{can\_raw}$ lead to a mapping $\mathcal{M}_{can} \equiv \mathcal{M}_{can\_raw}$, so $\mathcal{M}_{can} \equiv \mathcal{M}$. Thus, the set $\mathcal{E}$ is an ideal exemplar tuples set for $\mathcal{M}$.

□

Lemma 4.12. *Let $\mathcal{M}_{exp}$ be the mapping expected by the oracle Oracle.*
Let $\mathcal{E}$ be a set of exemplar tuples for $\mathcal{M}_{exp}$.
Let $\mathcal{M}_{can}$ be the canonical mapping computed from $\mathcal{E}$.
Let $\mathcal{E}_{ideal}$ be the ideal exemplar tuples set for $\mathcal{M}_{exp}$.
Let $\mathcal{Q} = \{q_1; ...; q_n\}$ be the following set of questions :

$$\{\text{“Is } E_S \text{ enough to deduce } E_T\text{?”} \mid (E_S, E_T) \in \mathcal{E}_{ideal}\}$$

Let $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_2} \ldots \xrightarrow{q_n} \mathcal{M}_{final}$ be an exploration over the set of questions $\mathcal{Q}$.
Then $\mathcal{M}_{final} \equiv \mathcal{M}_{exp}$

Proof. By construction of $\mathcal{E}_{ideal}$ (proof of Lemma 4.11), then *Oracle* will always answer `true` to each question in $\mathcal{Q}$. We also know by the construction of $\mathcal{E}_{ideal}$ that to each tgd in $\mathcal{M}_{exp}$ it corresponds to one, and only one, pair $(E_S, E_T) \in \mathcal{E}_{ideal}$.

Thus, each application of the transition rule $\mathcal{M}_i \xrightarrow{q} \mathcal{M}_{i+1}$ over a question $q \in \mathcal{Q}$ will add a tgd from $\mathcal{M}_{exp}$ to $\mathcal{M}_i$. At the end of the exploration over $\mathcal{Q}$, we obtain the mapping $\mathcal{M}_{final} = \mathcal{M}_{can} \cup \mathcal{M}_{exp}$. Thus, $\mathcal{M}_{final} \supseteq \mathcal{M}_{exp}$ and consequently $\mathcal{M}_{final} \models \mathcal{M}_{exp}$.

From Theorem 4.6, we also have that $\mathcal{M}_{exp} \models \mathcal{M}_{final}$, so $\mathcal{M}_{final} \equiv \mathcal{M}_{exp}$. □

Lemma 4.13. *Let $\mathcal{M}_{exp}$ be the mapping to describe (supposed in normal form).*
Let $\mathcal{E}_{FI}$ be a fully-informative exemplar tuples set *for $\mathcal{M}_{exp}$.*
Let $\mathcal{M}_{can}$ be the canonical mapping constructed from $\mathcal{E}_{FI}$.
Let $\mathcal{Q}$ be the set of questions asked from $\mathcal{M}_{can}$.
Let $\mathcal{E}_{ideal}$ be the ideal exemplar tuples set *for $\mathcal{M}_{exp}$ as described in Lemma 4.11.*

Then we have :

$$\mathcal{E}_{ideal} \subseteq \mathcal{Q}$$

i.e., our framework leads to explore the ideal exemplar tuples set $\mathcal{E}_{ideal}$.

Proof. For each tgd $(\phi \rightarrow \psi) \in \mathcal{M}_{exp}$ there exists an exemplar tuple (E_S, E_T) such that :

$$\exists E'_S \subseteq E_S \text{ s.t. } (\textsc{chase}(\sigma, E'_S) \neq \emptyset) \wedge (\textsc{chase}(\sigma, E'_S) \subseteq E_T)$$

By construction of $\mathcal{M}_{can}$, for each tgd $(\phi \rightarrow \psi) \in \mathcal{M}_{exp}$ there exists a tgd $(\phi' \rightarrow \psi') \in \mathcal{M}_{can}$ such that :

$$\exists \phi'' \subseteq \phi' \text{ s.t. } (\textsc{chase}(\sigma, \phi'') \neq \emptyset) \wedge (\textsc{chase}(\sigma, \phi'') \subseteq \psi)$$

So, there's a substitution μ such that all atoms in ϕ can be mapped to atoms in ϕ'' and an extension μ' of μ mapping all atoms of ψ to atoms in $\psi'' = \textsc{chase}(\sigma, \phi'')$. This lead to :

$$\mu(\phi) \subseteq \phi'' \subseteq \phi' \text{ and } \mu'(\psi) \subseteq \psi'' \subseteq \psi'$$

.

By construction of $\mathcal{E}_{ideal}$, for each tgd $(\phi \rightarrow \psi) \in \mathcal{M}_{exp}$ there exists $(E_S, E_T) \in \mathcal{E}_{ideal}$ such that $\phi = \bar{\theta}^{-1}(E_S)$ and $\psi = \bar{\theta}^{-1}(E_T)$. Thus, we can find a tgd $(\phi' \rightarrow \psi') \in \mathcal{M}_{can}$ such that there is a morphism μ and its extension μ' such that $\mu(\bar{\theta}^{-1}(E_S)) \subseteq \phi'$ and $\mu'(\bar{\theta}^{-1}(E_T)) \subseteq \psi'$. From this and from Definition 4.1 and Definition 4.2, for all exemplar tuples $(E_S, E_T) \in \mathcal{E}_{ideal}$ the question about the validity of tgd $\bar{\theta}^{-1}(E_S) \rightarrow \bar{\theta}^{-1}(E_T)$ is in the set $\boldsymbol{Q}$. □

Theorem 4.14 (Our framework can always produce a mapping logically equivalent to the expected one). *Let $\mathcal{M}_{exp}$ be the mapping expected by the oracle Oracle. Let $\mathcal{E}_{FI}$ be a fully informative exemplar tuples set for $\mathcal{M}_{exp}$. Let $\mathcal{M}_{can}$ be the canonical mapping computed from $\mathcal{E}_{FI}$. Let $\mathcal{M}_{can} \xrightarrow{q_1} \ldots \xrightarrow{q_n} \mathcal{M}_{final}$ be a complete exploration performed by our framework. Then :*

$$\mathcal{M}_{final} \equiv \mathcal{M}_{exp}$$

Proof. In Lemma 4.12 we show that if our framework ask all the questions of the ideal exemplar tuples set for the expected mapping $\mathcal{M}_{exp}$, then the output mapping $\mathcal{M}$ will be such that $\mathcal{M} \equiv \mathcal{M}_{exp}$. In Lemma 4.13 we show that, given a fully informative exemplar tuples set for $\mathcal{M}_{exp}$, then our framework will ask all questions of the ideal exemplar tuples set for $\mathcal{M}_{exp}$. From this follow our theorem. □

We will show now that the limitation over the pruning maintain the completeness of the framework.

Theorem 4.15. *Given an execution of our framework, the pruning does not affect the completeness of our approach.*

Proof. The pruning work in two ways :

- if $\mathcal{M}_{exp} \models \sigma$, then we prune each questions about a tgd σ' such that $\sigma \models \sigma'$. Trivially, there is no need to explore implied tgd of an already validated tgd as they can be validated by transitivity. Also, there is no need add them to the final mapping, as they can only create redundant tuples.
- if $\mathcal{M}_{exp} \not\models \sigma$, trivially we can prune each questions about a tgd σ' such that $\sigma' \models \sigma$.

□

5 EMBEDDING INTEGRITY CONSTRAINTS

In the previous section, we have described the core of our approach. Now, we will describe how a user can introduce integrity constraints to help the lattice pruning. Integrity constraints provide a way to define guidelines over a database schema, and ensure that the instances over this schema will comply to these guidelines. In practice, the most commonly used integrity constraints are primary keys and foreign keys (a particular case of inclusion dependencies). Such constraints are classic tools of database design, and therefore are easily available in real integration scenarios.

Introduction of integrity constraints constitutes an extension of our initial (IMS) problem. This *Interactive Mapping Specification with Integrity Constraints* approach (IMS_{IC}) can be stated as follows :

(IMS_{IC}) Given a set of exemplar tuples $\mathcal{E}$ and a (possibly empty) set of constraints Σ_{IC} provided by a non-expert user, given a mapping $\mathcal{M}$ that the user has in mind, the Interactive Mapping Specification with Integrity Constraints *problem is to discover, by means of boolean interactions, a mapping $\mathcal{M}'$ such that each $(E_S, E_T) \in \mathcal{E}$ satisfy $\mathcal{M}'$, $\mathcal{M}'$ is valid w.r.t. Σ_{IC} and $\mathcal{M}'$ generalizes $\mathcal{M}$.*

Considered schema	Primary keys	Foreign keys
Source	Not applicable	Section 5.1
Target	Section 5.2	Beyond scope

Fig. 5. Summary of the main theorems about our framework (in Section 4).

The possible cases of use of integrity constraints are summarized in Figure 5. Foreign keys over source schema and primary keys over target schema are addressed, respectively, in Section 5.1 and Section 5.2. Primary keys over the source schema cannot be used for our pruning, as this constraint should be satisfied by the source instances provided by users. Else, it will mean that user provided an instance which already violates the constraints over his schema, so this instance cannot be used to exemplify his schema mapping problem. The use of foreign keys over the target schema lead to non-trivial extension that are beyond the scope of this paper.

5.1 Use of foreign keys under source schema

The introduction of foreign keys constraints (and, more generally, of inclusion dependencies) can inform us about which tuple (containing a foreign key) can only occur in the presence of another tuple (referenced by the foreign key). This information between tuples can be represented by a graph as defined as follows :

Definition 5.1 (Foreign key constraint.). Let $\mathbf{R}$ be a database schema. Let S and T be two relation symbols such that $S, T \in \mathbf{R}$. Let X and Y be two distinct sequences of attributes over, respectively, S and T.

Then a foreign keys constraint is a constraint such that :

$$S[X] \subseteq T[Y] \text{ and } Y \text{ is a key of } T$$

Foreign keys are a particular case of *inclusion dependencies* constraints, for which Y don't need to be a key of T. Our system works with inclusion dependencies, so foreign keys are handled.

In our algorithm, we use *inclusion dependency graphs* to represent the constraints conveyed by the provided inclusion constraints over a conjunction of atoms. In such a graph, given each pair of atoms in the conjunction, there exists an directed edge between these atoms if they satisfy a provided inclusion constraint. More formally :

Definition 5.2 (Inclusion dependency graph.). Let ϕ be a conjunction of atoms over a schema $\mathbf{S}$. Let $\Sigma_{IC\_\mathbf{S}}$ be a set of integrity constraints over $\mathbf{S}$.

The inclusion dependency graph over ϕ is the directed graph :

$$\mathcal{G}_\phi = (atoms(\phi), E)$$

with :

$$E = \{\langle a_1, a_2\rangle \mid a_1 \in \phi, a_2 \in \phi, \exists \sigma \in \Sigma_{IC\_\mathbf{S}}, \langle \bar{\theta}(a_1), \bar{\theta}(a_2)\rangle \models \sigma\}$$

In this section, we make use of this graph during the atom refinement step as illustrated in the following example :

Example 5.3. Given two schemas $\mathbf{S} = \{S(x,y); U(x,y,z); V(z,x); W(z,x)\}$ and $\mathbf{T} = \{T(x)\}$. Given an exemplar tuple (E_S, E_T) over $\mathbf{S}$ and $\mathbf{T}$ such that :

$$E_S = \{S(\mathsf{a},\mathsf{b}), U(\mathsf{a},\mathsf{b},\mathsf{c}), V(\mathsf{c},\mathsf{a}), W(\mathsf{c},\mathsf{a}), S(\mathsf{d},\mathsf{e})\}$$

$$E_T = \{T(\mathsf{a})\}$$

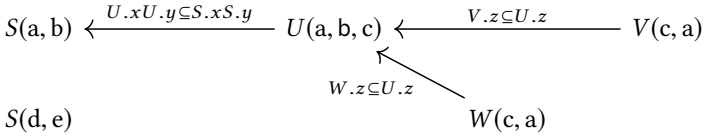


Fig. 6. Dependency graph of the atoms in ϕ_{E_S} (Example 5.3).

Given the corresponding conjunctions :

$$\phi_{E_S} = S(\mathrm{a}, \mathrm{b}) \wedge U(\mathrm{a}, \mathrm{b}, \mathrm{c}) \wedge V(\mathrm{c}, \mathrm{a}) \wedge W(\mathrm{c}, \mathrm{a}) \wedge S(\mathrm{d}, \mathrm{e})$$

$$\psi_{E_T} = T(\mathrm{a})$$

and the set of foreign keys over the source schema **S**:

$$\Sigma_{fk} = \{U.x, U.y \subseteq S.x, S.y; V.z \subseteq U.z; W.z \subseteq U.z\}$$

Then we can draw the dependency graph of the atoms in ϕ_{E_S} shown in Figure 6 (for the sake of clarity, edges are labeled with the corresponding inclusion dependency even if not used in our algorithm) :

We can see that $S(m, n)$ it is not linked to any other atom. At the opposite, atom $V(z, x)$ is linked to atom $U(x, z)$, and this last one is linked to atom $S(x, y)$. Therefore, we are sure that a tuple triggering atom $V(z, x)$ will always occur with tuples corresponding to atoms $U(x, z)$ and $S(x, y)$. As a consequence, we can skip exploring conjunctions like $V(z, x)$ and $U(x, z) \wedge V(z, x)$ during atom refinement step.

To make use of this, we propose Algorithm 4 in order to apply this optimization during atom refinement. In this algorithm, for the sake of clarity, we abuse the notation of $\mathcal{G}_\phi = (atoms(\phi), E)$ by simply writing $\mathcal{G}$ when it's obvious. To use it in Algorithm 1, the line

$$C_{cand} \leftarrow \textsc{SourceFk\_PruneUselessConjuction}(C_{cand}, C_{valid}, \Sigma_{sourceFk})$$

need to be inserted just after line 6.

This algorithm takes the set of candidate conjunctions that can be explored and prune it w.r.t. to inclusion dependencies. To achieve that, the algorithm begins by the construction of the dependency graph previously for each upper-bound of the quasi-lattices. Then, for each dependency graphs over an upper-bound, the algorithm check if the candidates that are subsets of this upper-bound respect all the dependencies of the graph. If such a candidate does not respect every dependency, it is pruned from the set of candidates output by the algorithm. In the following, we provide an example that substantiates the informal description of the algorithm.

Example 5.4 (Pruning of quasi-lattice : the need of evaluating each supremum separately.). Given two schemas $\mathbf{S} = \{S(x, y); S'(x, z); U(x, z)\}$ and $\mathbf{T} = \{T(x)\}$. Given two exemplar tuples over **S** and **T** such that :

$$(E_S^1, E_T^1) = (\{S(\mathrm{a}, \mathrm{b}), U(\mathrm{a}, \mathrm{c})\}, \{T(\mathrm{a})\})$$

$$(E_S^2, E_T^2) = (\{S'(\mathrm{a}, \mathrm{b}), U(\mathrm{a}, \mathrm{c})\}, \{T(\mathrm{a})\})$$

and the set of foreign keys over schema **S**:

$$\Sigma_{fk} = \{U.x \subseteq S.x; U.x \subseteq S'.x\}$$

During atom refinement, as these tgds are ψ-equivalent, we will explore the atoms sets quasi-lattice shown in Figure 7a

Algorithm 4 SourceFk_PruneUselessConjuction(C_{cand}, Σ_{fk})

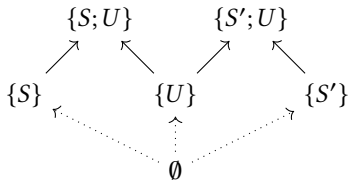
Input: A set C_{cand} of the candidate conjunctions to evaluate (as produced by Algorithm 1, line 5)
Input: A set C_{up} of the upper bound of the quasi lattice over C_{cand} (as produced by Algorithm 1, line 6)
Input: A set of foreign keys or other inclusion dependencies on the source schema Σ_{fk}.
Output: A set C'_{cand} of a the pruned set of candidates.

▷ Generation of dependency graphs for each upper-bound

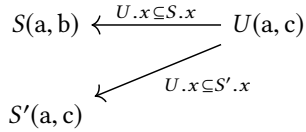
1: $\mathcal{F}_{\mathcal{G}} \leftarrow \emptyset$
2: **for all** $\phi_{up} \in C_{up}$ **do**
3: $E_{\phi_{up}} \leftarrow \emptyset$
4: **for all** $(R[X] \subseteq S[Y]) \in \Sigma_{fk}$ **do**
5: $E_t \leftarrow$ extract the pairs of atoms $\langle a_1, a_2 \rangle$ such that $a_1, a_2 \in \phi_{up}$ and $\bar{\theta}(a_1)[X] \subseteq \bar{\theta}(a_2)[Y]$
6: $E_{\phi_{up}} \leftarrow E_{\phi_{up}} \cup E_t$
7: **end for**
8: Let $\mathcal{G} = (atoms(\phi_{up}), E_{\phi_{up}})$
9: $\mathcal{F}_{\mathcal{G}} \leftarrow \mathcal{F}_{\mathcal{G}} \cup \{\mathcal{G}\}$
10: **end for**

▷ Pruning of candidates

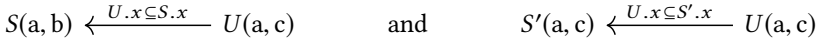
11: $C'_{cand} \leftarrow C_{cand}$
12: **for all** $\mathcal{G} \in \mathcal{F}_{\mathcal{G}}$ **do**
13: Let $\mathcal{G} = (atoms(\phi_{up}), E_{\phi_{up}})$
14: **for all** $c \in C'_{cand}$ such that $c \subseteq \phi$ **do**
15: **if** $\exists \langle a_1, a_2 \rangle \in E_{\phi_{up}}$ such that $a_1 \in c \wedge a_2 \notin c$ **then**
16: $C'_{cand} \leftarrow C'_{cand} \setminus c$
17: **end if**
18: **end for**
19: **end for**
20: **return** C'_{cand}



(a) Explored atoms sets quasi-lattice.



(b) Dependency graph without separate upper bound elements



(c) Dependency graphs when separate upper bound elements

Fig. 7. Explored quasi-lattice and dependency graphs of Example 5.4

If we don't produce separate dependency graphs for each element in the upper bound, we will obtain the graph in Figure 7b (for the sake of clarity, edges are labelled with the corresponding inclusion dependency even if not used in our algorithm): This graph will lead to the pruning of each conjunction except $S(a, b)$ and $S'a, c)$. This is due to the fact that the perfectly acceptable conjunction $S(a, b) \wedge U(a, c)$ and $S'(a, c) \wedge U(a, c)$ does not contain the whole set of dependencies expressed in the graph, and will be pruned by the condition line 15. In other words, this graph is only usable if the conjunction $S \wedge S' \wedge U$ can be accessed during atom refinement.

To avoid such a case, our algorithm constructs a dependency graph for each element in the upper-bound of the quasi-lattice. This allows to check, for each dependency graph of an element in the upper bound, if a candidate subset of this element does not express each of its dependencies. In our example, this lead to generate the two small dependency graphs shown in Figure 7c.

This graphs leads to prune conjunction $U(a, c)$ but not the conjunction $S(a, b) \wedge U(a, c)$ as it respects the dependency of the graph at the left and is not included in the other upper-bound element $S'(a, c) \wedge U(a, bc)$ (this prevents to evaluate this conjunction with the dependency graph at the right, which should have led to its pruning). The exact same principle leads to avoid the pruning of the conjunction $S'(a, c) \wedge U(a, c)$.

Lemma 5.5. *Given a quasi-lattice of atoms conjunctions as produced by our atom refinement step. Given the dependency graphs that are generated separately for each element e of the upper-bound of the quasi-lattice. If, for each graph of an element e, this graph is checked on a subset of e, then no valid conjunction will be suppressed.*

Proof. Given an element e and its corresponding graph $\mathcal{G}$, then our algorithm will suppress only candidates that are subsets of e and that violate at least one inclusion dependency represented in $\mathcal{G}$.

Moreover, given an atom $\delta \in e$ such that $e \setminus \delta$ violate an inclusion dependency in $\mathcal{G}$, this means that there is an atom $\gamma \in e$ such that there is an inclusion dependency from γ to δ, i.e. γ will always occur with the corresponding atom δ. Thus, there is no need to explore conjunction $e \setminus \delta$ as this conjunction will be triggered as often as conjunctions e.

□

5.2 Use of primary keys under target schema

During the steps of our framework, exploration can lead to evaluate tgds which can lead to break primary key constraints over the target schema as illustrated in the following example :

Example 5.6. Given exemplar tuples :

A	*Att1*	*Att2*	*Att3*
	a	b	a
	a	b	c
	c	b	c

$\rightarrow$

B	*Att4*	*Att5*	*Att6*
	a	b	a

The join refinement of variable a will explore the following possibilities :

$\sigma : A(a, b, a) \rightarrow B(a, b, a)$	$\text{CHASE}(\sigma, E_S) = \{B(a, b, a); B(c, b, c)\}$
$\sigma_1 : A(a, b, a') \rightarrow B(a, b, a')$	$\text{CHASE}(\sigma_1, E_S) = \{B(a, b, a); B(a, b, c); B(c, b, c)\}$
$\sigma_2 : A(a, b, a') \rightarrow B(a', b, a)$	$\text{CHASE}(\sigma_2, E_S) = \{B(a, b, a); B(c, b, a); B(c, b, c)\}$
$\sigma_3 : A(a, b, a') \rightarrow B(a, b, a)$	$\text{CHASE}(\sigma_3, E_S) = \{B(a, b, a); B(c, b, c)\}$
$\sigma_4 : A(a, b, a') \rightarrow B(a', b, a')$	$\text{CHASE}(\sigma_4, E_S) = \{B(a, b, a); B(c, b, c)\}$

Knowing that the pair of attributes $(B.Att4, B.Att5)$ is a primary key allow us to prune σ_1 and σ_2 as the result of chasing the source instance A with σ_1 and σ_2 will lead, respectively, to instances $\{B(a, b, a); B(a, b, c; B(c, b, c)\}$ and $\{B(a, b, a); B(c, b, c; B(c, b, c)\}$ which violate the constraint.

To handle that, we propose the Algorithm 5 which, given a set of primary key constraints provided by a user, allow to avoid exploration of candidates which can lead to break these constraints. To use it in algorithm 3, the condition line 8 needs to be changed by :

$(\nexists \sigma_t \in \Sigma_t, \sigma_t \models \sigma'') \wedge \textsc{AskJoinsValidity}(\sigma'') \wedge \neg \textsc{TargetPk\_InvalidTgd}(\sigma'', \Sigma, \{E_S^1; \ldots; E_S^n\})$

Algorithm 5 TargetPk_InvalidTgd($\sigma, \Sigma, \{E_S^1; \ldots; E_S^n\}$)

Input: A tgd σ to evaluate.
Input: A set of primary key constraints $\Sigma_{targetPk}$.
Input: A set of source instance $\{E_S^1; \ldots; E_S^n\}$ provided by the user (sources of the exemplar tuples and/or other sources).
Output: return `true` if the conjunction can be pruned, else return `false`.

1: **for all** $E_S^i \in \{E_S^1; \ldots; E_S^n\}$ **do**
2: **let** $E_T^i = \textsc{chase}(\sigma, E_S^i)$
3: $t_{bool} \leftarrow$ evaluate if E_T^i violates a primary key in $\Sigma_{targetPk}$
4: $res \leftarrow res \vee t_{bool}$
5: **end for**
6: **return** res

In the following lemma, we show that the introduction of our optimization over primary keys under target schema only prunes invalid candidates :

Lemma 5.7. *Given a candidate tgd during join refinement steps, then Algorithm 5 only prune invalid candidates.*

Proof. Our optimization lead to prune candidate tgds which will lead to violate the user's constraint if such tgds are applied to the user's examples. Thus, their invalidity is trivially seen. □

6 EXPERIMENTS

Our experimental study has three main objectives: (i) to provide a comparative analysis of the quasi-lattice approach with the semi-lattice approach proposed in [13], (ii) to evaluate the benefit of using exemplar tuples with respect to universal solutions for mapping refinement, and (iii) to provide a comparative analysis with [6].

Experimental settings. We have implemented our framework using OCaml 4.03 on a 2.6GHz 4-core, 16Gb laptop running Debian 9. We have borrowed mappings from seven real integration scenarios of the `iBench` benchmark [8]. The left part of Table 1 reports the *size* of each considered mapping scenario as the total number of tgds ($|\Sigma|$) and the average number of occurrences of their variables ($\bar{N}$) defined as $\bar{N} = \Sigma_{v \in V} N_v / |V|$, with V being the set of distinct variables and N_v the number of occurrences of each v variable within the tgds. Since there exists a bijection between variables and constants, $\bar{N}$ also stands for the average number of occurrences of constants per instance in the exemplar tuples.

Methodology. In all experiments, we consider the `iBench` mapping scenarios as the ideal mappings that the user has in mind. Starting from these mapping scenarios, we construct exemplar tuples as follows. Each tgd $\sigma \in \Sigma$ of the form $\phi \rightarrow \psi$ is transformed into a pair of instances $({E_S}^\sigma, {E_T}^\sigma)$, ${E_S}^\sigma$ (${E_T}^\sigma$, resp.) being generated by replacing each atom in ϕ (ψ, resp.) by its tuple counterpart with freshly picked constants for each variable in the tgd. Thus, for each scenario $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, we obtain a set of exemplar tuples $E_\Sigma = \{({E_S}^{\sigma_1}, {E_T}^{\sigma_1}), \ldots, ({E_S}^{\sigma_n}, {E_T}^{\sigma_n})\}$.

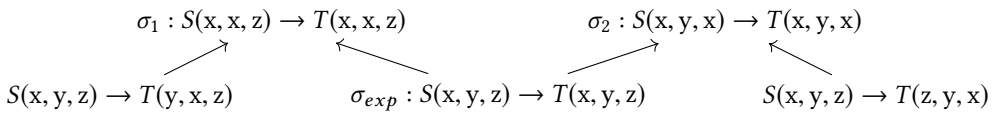


Fig. 8. Quasi-lattice of Example 6.1

These exemplar tuples are used as a baseline in our experimental study, as we expect that an "ideal" user, who does not make any mistakes, would actually produce such examples. In order to introduce user ambiguities in the above tuples, we have built alternative test cases, in which the exemplar tuples E_Σ are degraded . The degradation procedure is meant to reproduce users' common mistakes while specifying exemplar tuples.

As opposed to [13], we only focus here on atom degradation. The reason is that the introduction of quasi-lattice does not add much improvement to the join refinement step, unless in some specific cases as illustrated the following example :

Example 6.1. Let the expected mapping be :

$$\Sigma_{exp} = \{\sigma_{exp} : S(\mathrm{x}, \mathrm{y}, \mathrm{z}) \rightarrow T(\mathrm{x}, \mathrm{y}, \mathrm{z})\}$$

Let the exemplar tuples provided by the user be :

$$(E_{S1}, E_{T1}) = (\{S(\mathsf{a}, \mathsf{a}, \mathsf{c})\}, \{T(\mathsf{a}, \mathsf{a}, \mathsf{c})\})$$
$$(E_{S2}, E_{T2}) = (\{S(\mathsf{a}, \mathsf{b}, \mathsf{a})\}, \{T(\mathsf{a}, \mathsf{b}, \mathsf{a})\})$$

Then this set of exemplar tuples lead to the following canonical mapping :

$$\Sigma_{can} = \{\sigma_1 : S(\mathrm{x}, \mathrm{x}, \mathrm{z}) \rightarrow T(\mathrm{x}, \mathrm{x}, \mathrm{z});$$
$$\sigma_2 : S(\mathrm{x}, \mathrm{y}, \mathrm{x}) \rightarrow T(\mathrm{x}, \mathrm{y}, \mathrm{x})\}$$

Hence, the occurrences of the constant a to represent variable y in (E_{S1}, E_{T1}) and variable z in (E_{S2}, E_{T2}) lead to explore the quasi-lattice of tgds shown in Figure 8 during join refinement of x.

For our comparison with the approach used in [13], we have not created a new degradation method to artificially improve our results. We keep the method in [13] which is more realistic and give us a fair comparison.

Atom degradation procedure is parametrized by the total number of extraneous tuples added to E_Σ, with the constraint that at most one tuple is added to each individual instance ${E_S}^{\sigma_i}$. An extraneous tuple is generated by randomly choosing a source instance ${E_S}^{\sigma_i}$, picking a tuple at random within it, copying it and then replacing one constant of the tuple with a fresh one.

Example 6.2. By applying the degradation procedure on the tgd σ from Example 3.20, the following exemplar tuples may be yielded $({E_S}'^{\sigma}, {E_T}'^{\sigma})$. An extraneous F' atom is added, the degradation being underlined:

$${E_S}'^{\sigma} = \{F(\mathtt{f0}, \mathtt{Miami}, \mathtt{L.A.}', \mathtt{a0}); \underline{F'(\mathtt{f0}, \mathtt{Miami}, \mathtt{L.A.}', \mathtt{a0})}; A(\mathtt{a0}, \mathtt{AA}, \mathtt{L.A.}'')\}$$
$${E_T}'^{\sigma} = \{Dpt(\mathtt{L.A.}', \mathtt{f2}, \mathtt{c0}); Arr(\mathtt{L.A.}', \mathtt{f2}, \mathtt{c0}); Co(\mathtt{c0}, \mathtt{AA}, \mathtt{L.A.}'')\}$$

In our experimental study, we have deteriorated each initial set of examples E_Σ by adding 0, 2, 5, 8 or 10 tuples.

For each of the above configurations, we repeated the degradation procedure 30 times in order to obtain an equivalent number of degraded test cases.

Moreover, we simulate the user's answers during the interactive part of our approach with the following assumption: the user always replies correctly to a given challenge (i.e. an input pair

(E_S, E_T)) w.r.t. the original mapping Σ from the scenario. In order to simulate the user answer, E_S is chased to obtain $E_T{}'$. 'Yes' is produced as an answer if there exists a substitution μ from E_T into $E_T{}'$ such that $\mu(E_T) \subseteq E_T{}'$, otherwise 'No' is returned.

Benefit of quasi-lattices. In the first experiment, we gauge the effectiveness of using quasi-lattices structures compared to the isolated semi-lattices used in [13]. We focus on the Breadth-First exploration strategies, both in Top-Down and Bottom-Up versions, as they've been shown to be the more efficient in [13]. The use of quasi-lattices show to have a statistically significant correlation with the number of questions asked during atom refinement ($p - value = 4.74e - 14$, tested with a `MANOVA`). In the following, we will analyse the results of our experiments.

Table 1 presents the reduction obtained by the use of quasi-lattices (in percent) over the average number of questions per tgd asked to the user ($\Delta\bar{x}$), and over the maximum number of questions per tgd (Δx_m) for a scenario. It can be seen that the reduction of the average number of questions by the use of quasi-lattices range from 0% to 12.7%, with a global reduction of 3.2%. Also, the reduction of the maximal number of questions by the use of quasi-lattices range from 0% to 27%, with a global reduction of 11.8%.

The efficiency of the optimisation is not directly correlated with the number of tgds in a scenario. This is illustrated with scenarios `amalgam2` and `SDB1-to-SDB3`, where the biggest one (`amalgam2`) lead to a small amelioration, when the other one lead to the highest reduction of the number of questions asked. This can be explained by the structure of the tgds contained by the scenarios. When scenarios contain numerous but non overlapping tgds (i.e., our degraded exemplar tuples sets lead to few ψ-equivalent tgds), most of the quasi-lattices cover one tgd at a time and consequently are equivalent to the semi-lattices used in [13]. In the other case, even with less tgds than in `amalgam2`, the use of quasi-lattices during refinement of scenario `SDB1-to-SDB3` lead to an important reduction of the number of question asked because this scenario contains tgds which are differentiated by more subtle differences than in `amalgam2`. Such scenarios with numerous ψ-equivalent tgds leads to exemplar tuples sets which are efficiently handled by the use of quasi-lattices.

Figure 9 illustrates the number of questions asked for each configuration. We recall that, in a boxplot : the central bar correspond to the median; the lower and upper edges are, respectively, the first and fourth quartiles; the lower and upper whisker are, respectively, the minimum and maximum values excluding outliers; the isolated points are the outliers.

The boxplots shown that in the worst case values stay unchanged, and in the scenarios with the highest number of questions asked (such as `SDB1-to-SBD3` and `amalgam2`) the use of quasi-lattice leads to efficiently decrease the number of questions asked. This is shown by the decrease of the fourth quartiles value (i.e., the value such that 75% of data have a lower value) and by the suppression (complete or partial) of outliers occurrences.

It is worth to note that, in some cases such as the scenario `GUS-to-BIOSQL` with 2 degradations in Figure 9.(d), the use of quasi-lattices lead to a greater upper whisker for the same degraded scenarios. This is due to the fact that upper whisker does not consider outliers. Hence, for degraded scenarios leading to outliers without the use of quasi-lattices, the reduction of the number of questions with the use of quasi-lattice lead to datapoints that are no more classified as outliers and increase the upper whisker value. This can be seen in Figure 9 on scenario GUS-to-BIOSQL with 2 degradations, In this scenario, the case without quasi-lattices leads to outliers values up to 16 question, when the use of quasi-lattice. over the same degraded scenario reduce the number of interactions to a maximum of 13 questions (which is not an outlier value).

Benefit of (non-universal) exemplar tuples. Our second experiment aims to evaluate the benefit of using exemplar tuples as opposed to universal examples adopted in [6] for the mapping inference process. For each scenario, we apply the chase to all the source instances E_S^i to obtain

Scenarios				Reduction obtained with use of quasi-lattice	
name	Degradations	$\|\Sigma\|$	$\bar{N}$	$\Delta\bar{x}$	Δx_m
a1-to-a2	0	8	2.5	0%	0%
	2	8	2.5	3%	13.3%
	5	8	2.5	4.2%	16%
	8	8	2.5	5.6%	17.2%
amalgam2	0	71	1.3	0%	0%
	2	71	1.3	0.5%	9.1%
	5	71	1.3	1%	19.9%
	8	71	1.3	0.8%	8.8%
	10	71	1.3	0.9%	7.7%
dblp-amalgam	0	10	1.4	0%	0%
	2	10	1.4	0.2%	12.5%
	5	10	1.4	0.8%	14.3%
	8	10	1.4	1%	6.7%
	10	10	1.4	1.6%	8.7%
GUS-to-BIOSQL	0	8	1.5	0%	0%
	2	8	1.5	1.7%	18.75%
	5	8	1.5	2.3%	15%
	8	8	1.5	2.2%	11.5%
SDB1-to-SDB2	0	10	1.5	0%	0%
	2	10	1.5	1.7%	15.8%
	5	10	1.5	3.2%	16%
	8	10	1.5	4.6%	16.6%
	10	10	1.5	5.1%	14.7%
SDB1-to-SDB3	0	11	1.5	12.7%	14.3%
	2	11	1.5	12.4%	20%
	5	11	1.5	12.1%	22%
	8	11	1.5	12.4%	26.9%
	10	11	1.5	11.6%	27%
SDB2-to-SDB3	0	9	2.1	0%	0%
	2	9	2.1	0.4%	12.5%
	5	9	2.1	1.2%	9.1%
	8	9	2.1	1.4%	11.1%
	10	9	2.1	1.6%	9.5%
Mean				3.2%	11.8%

Table 1. Scenarios characteristics; reduction of the number of asked questions obtained with use of quasi-lattices over average ($\Delta\bar{x}$) and maximum (Δx_m) number of questions per tgd and for each dataset.

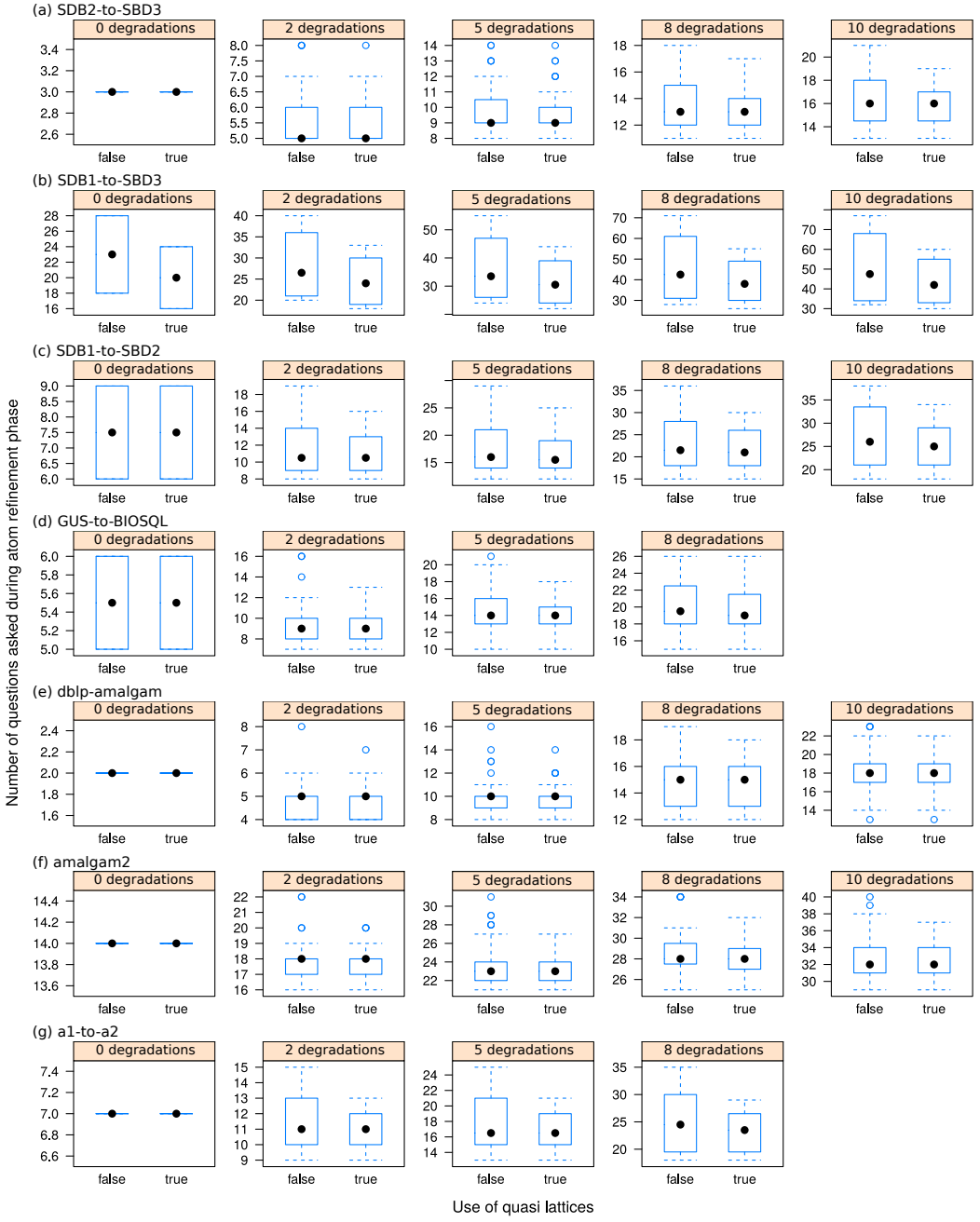


Fig. 9. Comparison of the number of questions per tgd asked during atom refinement step, with and without use of the quasi-lattices, from 0 to 10 atom degradations.

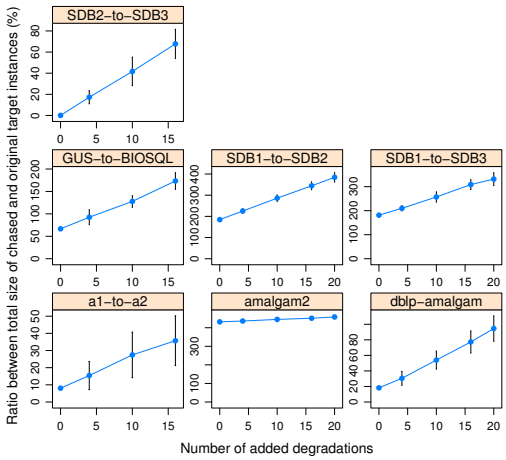


Fig. 10. Growth of the ratio r wrt. number of degradations.

$\text{CHASE}(\mathcal{M}, E_S^i)$. This lets us compute the number of universal exemplar tuples, which we compare with the number of targets (non-universal) exemplar tuples used in our approach. Concretely, for exemplar tuples $\{(E_S^1, E_T^1); \ldots; (E_S^n, E_T^n)\}$ and the corresponding expected mapping $\mathcal{M}$, we calculate the ratio $r = \frac{\Sigma_{i=1}^n |\text{CHASE}(\mathcal{M}, E_S^i)|}{\Sigma_{i=1}^n |E_T^i|} - 1$ of additional atoms in the generated solution. In order to get a comprehensive view of the effects of atom and join degradations, both degradations occur together in this experiment. Precisely, in Figure 10, we present the results where an equal number of atom and join degradations are used. The x axis corresponds to the total number of degradations (e.g., the value 20 corresponds to the case with 10 atoms and 10 join degradations), while the y axis corresponds to the aforementioned ratio r.

In all the employed scenarios, we can observe the effectiveness and practicality of using exemplar tuples as opposed to the universal data examples of EIRENE: universal exemplar tuples are from 30% to 458% larger than the non-universal ones used in our approach. Moreover, in all scenarios, we can observe a strong linear correlation between the number of degradations and the number of additional target tuples needed by universal examples. Hence, the more degradations the exemplar tuples have, the larger is the benefit of using our approach. Notice that the scenario that is the less sensitive to the variation of the number of degradations is amalgam2, which is also the scenario with the greatest number of tgds. Such a scenario is also among those that exhibited the maximum benefit of using fewer exemplar tuples rather. Although the precise amount of gain is clearly dependent on the dataset and on the number of degradations, *we can observe that, in all scenarios, the advantage of using non-universal exemplar tuples is non-negligible, thus making our approach a practical solution for mapping specification.*

Relative benefit of interactivity. A key contribution of our mapping specification method is that it helps the user to interactively correct errors (e.g., unnecessary atoms during atom refinement, collisions of constants during join refinement) that may appear in the exemplar tuples. In this section, we aim at quantifying this benefit via a comparison with a baseline approach, i.e., the one in which refinement steps are disabled. As a baseline, we adopted the canonical GLAV generation performed in EIRENE[3]. As EIRENE is not intended to handle errors in its input data examples, we

[3]For the sake of fairness, EIRENE's canonical GLAV are *split-reduced* and σ*-redundant* tgds are suppressed.

Scenarios	Number of extraneous atoms added				
	0	2	5	8	10
SDB2-to-SDB3	0	12.4	27.0	36.9	-
SDB1-to-SDB2	0	11.1	23.8	33.3	38.5
SDB1-to-SDB3	0	7.3	16.2	23.6	28.0
dblp-amalgam	0	12.2	25.3	35.5	40.7
GUS-to-BIOSQL	0	11.7	25.8	35.7	-
a1-to-a2	0	8.3	18.5	26.6	-
amalgam2	0	3.1	7.5	10.9	12.8
Average	0	9.5	20.6	28.9	30

Table 2. Relative difference (in percent) between EIRENE and our system.

had to make sure that exemplar tuples in our case are an acceptable input for EIRENE, in particular that they pass the so-called "homomorphism extension test". In other words, we bootstrap our algorithms on universal exemplar tuples (E_S, E_T) in order to warrant such comparison.

We use the sum of the number of left-hand side atoms of the tgds as the comparison criterion: the larger it is, the more "complex" is the mapping for the end user. This optimality criterion is inspired by a compound measure proposed in [23]. Notice that this comparison only deals with extraneous atoms during atom refinement and does not consider collision of values, which is done during join refinement. For such a reason, and also due to the fact that here we are compelled to use universal data examples instead of few arbitrary exemplar tuples in order to compare with EIRENE, this comparison should be taken with a grain of salt.

The obtained results are presented in Table 2. If no extraneous atom is added to the left-hand sides of mappings, then there is no qualitative difference between the two approaches. However, when extraneous atoms are introduced, a remarkable difference can be observed: EIRENE's canonical mapping is about 20% larger on average (across all scenarios) when 5 such atoms are introduced, and goes up to 30% on average with 10 atoms. *Hence, our mappings are noticeably simpler than* EIRENE*'s ones. Such an improvement is both beneficial for the readability of mappings as well as for their efficiency because spurious atoms are eliminated.*

7 RELATED WORK

A pioneering work on the usage of data examples in mapping understanding and refinement [33] relies on Clio's [28] schema correspondences as specified in a graphical user interface. By leveraging such correspondences, Yan et al. [33] propose alternative data associations among relevant source instances leading to construct mappings in an incremental fashion with the intervention of the mapping designer. The dichotomy between the expected user instance and the generated instance has been further investigated in Routes [17]. The input required by Routes consists of both a source instance and a mapping that the user readily intends to debug. The user then builds test cases for the mapping at hand by probing values in the target instance, and the system returns a provenance trace to explain how and why the probed values are computed. This approach closely resembles testing as done for software development. By opposite, our method requires as inputs a source and target exemplar tuples and no prior mapping connecting them. The final objective of our approach, which especially targets users unfamiliar with schema mappings, is to build the mapping that the user had in mind via simple boolean user interactions. To draw a comparison with software

development, our method generates a specification (i.e. a mapping expressed in first-order logic) starting solely from supplied unit test cases (i.e. small exemplar tuples).

As in Yan et al. [33], Muse [4] leverages data examples to differentiate between alternative mapping specifications of the designer and drives the mapping design process based on the designer's actions. However, the techniques proposed in [4] are more sophisticated than the ones in [33], in that they address the problem of the grouping semantics of mappings and their alternative semantics in case of ambiguity. Muse also poses a number of yes/no questions to the designer to clarify the grouping semantics. However, the number of questions are driven by the schema elements along with schema constraints that are used to reduce the number of questions. In our approach, we do not assume prior knowledge of the schema constraints. TRAMP [21] and Vagabond [22] focus on the understandability of user errors in mappings by using provenance. However, explanations returned by Vagabond are to be interpreted by users who are familiar with the mapping language and its underlying semantics.

The use of data examples as evaluation tools has begun in [3, 31], which investigated the possibility of uniquely characterizing a schema mapping by means of a set of data examples. Hence, such unique characterization, up to logical equivalence of the obtained mappings, using a finite set of universal data examples was shown to be possible only in the case of LAV dependencies and for fragments of GAV dependencies [3, 31]. As a negative result, it was shown in [3] that already simple s-t tgds mappings, such as copy $E(x, y) \rightarrow F(x, y)$, cannot be characterized by a finite set of universal data examples under the class of GLAV mappings. Given the impossibility of uniquely characterizing GLAV mappings in real settings, [5, 6] made the choice of being less specific. Precisely, they decided to characterize, for a given schema mapping, the set of valid "non-equivalent" mappings with respect to the class of GLAV. To achieve that, they rely on the notion of "most general mapping". It was shown that, given a schema mapping problem, a most general mapping always exists in the class of GLAV mappings if there exists at least one valid mapping for the considered problem [5].

In EIRENE [6], the authors show how the user can generate a mapping that fits universal data examples given as input. Whereas EIRENE expects a set of *universal* data examples, we lift the universality assumption arguing that universal data examples are hard to be produced by a non-expert user. Moreover, as we have shown in Section 6, universal target instances tend to be *significantly larger* than our exemplar tuples. One previous work targeting non-expert users is MWeaver [29], where the user is asked to toss tuples in the target instance by fetching constants within the available complete source instance. However, this work has different assumptions with respect to ours: it aims at searching a source sample among all possible samples satisfying the provided target tuples, focusing on GAV mappings only. Our system inspects a few input tuples, on which interactive refinement is enabled, and expressive GLAV mappings can be inferred via simple user feedback. As mentioned in the introduction, we significantly improve the approach in [13] by providing formal guarantees of the quality of the obtained mappings, a more efficient data structure to explore the search space and the adoption of integrity constraints as metadata in order to reduce the number of user interactions.

All the aforementioned approaches are meant to produce the best exact mapping. However, one can use data examples to produce approximate mappings. Gottlob and Senellart propose a cost-based method to estimate the best approximate mapping given a set of possible repairs of the initial mapping [24]. The cost function takes account for the length of the generated tgds and the number of *repairs* that are needed to obtain a tgd that fully explain the instance E_T. Approximation of schema mappings has been considered recently in [15] by considering more expressive fragments of GLAV and GAV.

Cate et al. [14] show how computational learning (i.e., the *exact learning model* introduced by D. Angluin [7] and the Probably Approximately Correct model introduced by L. Valiant [32]) can be used to infer mappings from data examples. Their analysis is restricted to GAV schema mappings. Recently, Cate et al.[16] have employed active learning to learn GAV mappings and proved its utility in practice.

Besides mapping specification and learning, researchers have investigated the problem of inferring relational queries [1, 2, 12, 27]. The work in [1, 2] focuses on learning quantified Boolean queries by leveraging schema information under the form of primary-foreign key relationships between attributes. Their goal is to disambiguate a natural language specification of the query, whereas we use raw tuples to guess the unknown mapping that the user has in mind. In [12], the problem of inferring join predicates in relational queries is addressed. Consistent equi-join predicates are inferred by questioning the user on a unique denormalized relation. We differ from their work as follows: we focus on mapping specification and consider the broad class of *GLAV mappings* whereas they focus on query specification for a limited fragment of (equi-join) queries. Finally, [27] presents the exemplar query evaluation paradigm, which relies on exemplar queries to identify a user sample of the desired result of the query and a similarity function to identify database structures that are similar to the user sample. For the latter, the input database is assumed to be known, which is not an assumption in our framework. Since exemplar queries are answered upon an input database, they are considered as unambiguous, whereas this is not necessarily the case in our framework, whose goal is to refine and disambiguate exemplar tuples to derive the unknown mapping that the user has in mind.

8 CONCLUSIONS

We have addressed the problem of interactive schema mapping inference starting from arbitrary sets of exemplar tuples, as provided by non-expert users. We have shown that simplification of the mappings is possible by alternating normalization and refinement steps, the latter under the form of simple boolean questions. Compared to [13], we provide more tight formal guarantees, quasi-lattices to explore the space of possible mappings and the adoption of integrity constraints in order to reduce the number of questions that need to be asked to the user.

This paper lays the foundations of a practical framework that makes data exchange feasible for the masses. Much work is left to be done in order to make mapping specification an activity for non-expert users, for instance by adding features like error acceptance in user responses. Different gradients of users with more or less expertise can be captured with user modeling, which is beyond the scope of our work. Another future direction is to leverage machine learning techniques, such as Inductive Logic Programming, to automatically explore the space of mappings and confront their results with those obtained in our framework.

REFERENCES

[1] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *Proceedings of PODS*, pages 49–60, 2013.

[2] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12):1938–1941, 2012.

[3] B. Alexe, B. T. Cate, P. G. Kolaitis, and W.-C. Tan. Characterizing schema mappings via data examples. *TODS*, 36(4):23:1–23:48, 2011.

[4] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *Proceedings of the ICDE*, pages 10–19, 2008.

[5] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *Proceedings of SIGMOD*, pages 133–144, 2011.

[6] B. Alexe, B. Ten Cate, P. G. Kolaitis, and W.-C. Tan. Eirene: Interactive design and refinement of schema mappings via data examples. *Proceedings of VLDB*, 2011.

[7] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[8] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *Proceedings of VLDB*, 9(3):108–119, 2015.

[9] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[10] Z. Bellahsene, A. Bonifati, and E. Rahm, editors. *Schema Matching and Mapping*. Data-Centric Systems and Applications. Springer, 2011.

[11] P. A. Bernstein and S. Melnik. Model management 2.0: Manipulating richer mappings. In *SIGMOD*, 2007.

[12] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, Jan. 2016.

[13] A. Bonifati, U. Comignani, E. Coquery, and R. Thion. Interactive mapping specification with exemplar tuples. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 667–682, New York, NY, USA, 2017. ACM.

[14] B. T. Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM TODS*, 38(4):28, 2013.

[15] B. T. Cate, P. G. Kolaitis, K. Qian, and W.-C. Tan. Approximation Algorithms for Schema-Mapping Discovery from Data Examples. *ACM Trans. Database Syst.*, 42(2):12:1–12:41, 2017.

[16] B. T. Cate, P. G. Kolaitis, K. Qian, and W.-C. Tan. Active learning of gav schema mappings. In *Proceedings of PODS*, 2018.

[17] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. In *Proceedings of the 32nd international conference on Very large data bases*, pages 79–90. VLDB Endowment, 2006.

[18] G. I. Diaz, M. Arenas, and M. Benedikt. Sparqlbye: Querying RDF data by example. *PVLDB*, 9(13):1533–1536, 2016.

[19] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[20] M. J. Franklin, A. Y. Halevy, and D. Maier. A first tutorial on dataspaces. *PVLDB*, 1(2):1516–1517, 2008.

[21] B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. Tramp: Understanding the behavior of schema mappings through provenance. *Proc. VLDB Endow.*, 3(1-2):1314–1325, Sept. 2010.

[22] B. Glavic, J. Du, R. J. Miller, G. Alonso, and L. M. Haas. Debugging data exchange with vagabond. *PVLDB*, 4(12):1383–1386, 2011.

[23] G. Gottlob, R. Pichler, and V. Savenkov. Normalization and optimization of schema mappings. *VLDB J.*, 20(2):277–302, 2011.

[24] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *Journal of the ACM*, 57(2):6, 2010.

[25] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of SIGMOD*, pages 13–24, 2007.

[26] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. 4(4):455–469.

[27] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[28] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, and R. Fagin. Translating web data. In *Proceedings of VLDB*, pages 598–609, 2002.

[29] L. Qian, M. J. Cafarella, and H. Jagadish. Sample-driven schema mapping. In *Proceedings of SIGMOD*, pages 73–84. ACM, 2012.

[30] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, pages 146–171, 2005.

[31] B. Ten Cate, P. G. Kolaitis, and W.-C. Tan. Database constraints and homomorphism dualities. In *CP*. Springer, 2010.

[32] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, Nov. 1984.

[33] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proceedings of SIGMOD*, pages 485–496, 2001.