On Keys, Foreign Keys and Nullable Attributes in Relational Mapping Systems

Luca Cabibbo

RT-DIA-138-2008 December 2008
ABSTRACT

We consider the following main scenario for a mapping system: given a source schema, a target schema, and a set of value correspondences between these two schemas, visually specified as lines, generate an executable transformation (i.e., a set of queries) to compute target instances from source instances. We base this computation on two main components: (i) a schema mapping generation algorithm, to compute a declarative schema mapping from the correspondences, and (ii) a query generation algorithm, to compute a transformation from the schema mapping. Specifically, we introduce novel schema mapping and query generation algorithms for mappings between relational schemas with keys, foreign keys and nullable attributes. We extend current relational mapping algorithms (e.g., those proposed in the Clio framework), which are able to deal only in a more limited way with such integrity constraints. As a further contribution, we propose referenced-attribute correspondences, which permit to specify more precise mappings than traditional attribute correspondences, while retaining a simple and intuitive semantics.
1 Introduction

A common need in many application contexts is to transform and exchange data stored under different representations or schemas [3, 7]. A mapping is a precise specification that describes the relationship between two database schemas, a source and a target schemas. Recently, many mapping systems have been developed to cope with the difficulties of designing mappings [3, 18], including both research prototypes, such as Clio [14, 16], and commercial industry tools, such as Altova MapForce, IBM Rational Data Architect, Microsoft BizTalk Mapper, and Stylus Studio XML Mapper. \(^1\)

We consider the following main scenario for a mapping system: Given a source schema and a target schema, and a high-level specification of the value correspondences between elements of these two schemas (correspondences can be depicted as lines in a visual interface), compute an executable transformation from the source to the target schema (e.g., a set of queries to compute target instances from source instances). This computation can be performed using an intermediate step, as in Clio [16]: first, a schema mapping is computed from the value correspondences; then, a transformation is computed from the schema mapping. A schema mapping is a declarative specification of the mapping, which can be expressed using, in particular, source-to-target tuple-generating dependencies [5]. This way, we have identified the two main components of a mapping system we are interested in: the schema mapping generation algorithm and the query generation algorithm. (A mapping system may have further components, e.g., a matching algorithm to automatically discover correspondences between the source and target schemas. However, the focus of this paper is only on schema mapping and query generation algorithms.)

Schema mapping and query generation algorithms, initially developed in the Clio project to cope with relational schemas [14], have then been extended to deal also with XML nested data [16]. Further research has then mainly focused on improving the translation between XML data (e.g., [6, 18]). Many results in the nested setting can be applied to the flat relational case as well. However, the relational context has not been fully explored yet. In particular, many proposals take into account, separately, different integrity constraints that are common in the relational context, such as keys (e.g., [19]), foreign keys (e.g., in the form of inclusion dependencies [14, 16]), and nullable attributes (e.g., [16, 19]) but, to the best of our knowledge, a comprehensive approach to deal with all these constraints together has not been proposed yet.

The main contribution of this paper is the definition of a novel schema mapping generation algorithm and a novel query generation algorithm for relational mapping systems, for managing keys, foreign keys and nullable attributes in a comprehensive way. Intuitively, our schema mapping generator takes into account nullable attributes, by generating a rich set of logical mappings with null and non-null conditions. Then, our query generator deals with target key constraints, by rewriting and combining logical mappings intended to propagate data over a same target relation. We show, by mean of examples, that our algorithms compute “more desirable” mappings and transformations than current algorithms; by “more desirable” we mean with a more natural semantics (closer to the canonical universal instance semantics of [19, 5]) and with fewer useless tuples in target instances (i.e., partially duplicate tuples or tuples containing only null or invented values).

A further contribution of this paper is the introduction of referenced-attribute correspondences, which generalize traditional attribute correspondences [14, 16]. They permit to express more precise mappings, while retaining a simple and intuitive semantics.

Our algorithms can be applied to mapping problems involving relational schemas with primary keys, foreign keys (used to reference simple keys and forming a weakly acyclic set [5]), and nullable attributes. As output, our algorithms generate transformations expressed as non-

recursive Datalog queries [1], with Skolem functors [8] and safe (stratified) negation [1].

**Organization of the Paper.** Section 2 presents a number of motivating examples. Section 3 recalls preliminary notions, including known basic schema mapping and query generation algorithms. We introduce referenced-attribute correspondences in Section 4. We then present our novel schema mapping and query generation algorithms in Sections 5 and 6, respectively. Finally, Sections 7 and 8 are devoted to related work and conclusions, respectively.

This paper is an extended version of [4]. This paper contains a number of detailed definitions, procedures, and examples that have been omitted in [4] for space reasons. These parts are usually written inside the main body of the paper. Moreover, Appendix A proposes the motivation for our choices about the management of nullable attributes, Appendix B discusses a number of skolemization procedures, and Appendix C is devoted to additional and more comprehensive examples.

## 2 Motivating Examples

Our first example shows benefits that can be gained by managing keys, foreign keys and nullable attributes in relational mappings.

**Example 2.1** Consider the mapping problem graphically depicted in Figure 1. The schemas comprise keys (shown underlined), foreign keys (shown by means of dashed arrows), and nullable attributes (shown using the $null$ superscript). The mapping is specified as a set of attribute correspondences, visually depicted as solid arrows. It involves two car registration databases: $CARS_3$, the source schema, shown on the left, and $CARS_2$, the target schema, shown on the right. Attributes $person$ and $car$ represent person and car identifiers (e.g., SSN’s and plates), respectively.

For this mapping problem, basic schema mapping generation algorithms [14, 16] (which do not fully consider nullable attributes) compute the following schema mapping (for space reasons, we denote relation and attribute names only by their initial letters):

$$
P_3(p, n, e) \rightarrow P_2(p, n, e)
$$

$$
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
$$

$$
C_3(c, m) \rightarrow C_2(c, m, p'), P_2(p', n', e')
$$

The third of these logical mappings is undesirable, since it intuitively states that each car has an owner, while in both schemas there can be cars without an owner.
Furthermore, basic query generation algorithms [14, 16] (which don’t consider key constraints) compute the following transformation from $CARS_3$ to $CARS_2$ (expressed as a non-recursive Datalog program with Skolem functors):

\[
\begin{align*}
P_2(p, n, e) & \leftarrow P_3(p, n, e) \\
C_2(c, m, p) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \\
C_2(c, m, f_P(c, m)) & \leftarrow C_3(c, m) \\
P_2(f_P(c, m), f_N(c, m), f_E(c, m)) & \leftarrow C_3(c, m)
\end{align*}
\]

While the first two rules correctly deal with persons and cars having an owner, the last two rules incorrectly deal with all cars, by “inventing” a new owner (a person) for each car, even for cars already having a known owner.

While the first two rules correctly deal with persons and cars having an owner, the last two rules incorrectly deal with all cars, by “inventing” a new owner (a person) for each car, even for cars already having a known owner.

The above example suggests the need for schema mapping and query generation algorithms that take into account a wider set of integrity constraints than current solutions.

Example 2.1 (cont.) A more natural and desirable data transformation for the mapping problem of Example 2.1 is the one shown in Figure 3.

For the same mapping problem, our novel schema generation algorithm (which also considers nullable attributes) computes the following schema mapping:

\[
\begin{align*}
P_3(p, n, e) & \rightarrow P_2(p, n, e) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow C_2(c, m, p), P_2(p, n, e) \\
C_3(c, m) & \rightarrow C_2(c, m, p')
\end{align*}
\]

Here, the second logical mapping deals with owned cars, while the third logical mapping deals with all cars, with or without an owner. As we shortly see, a null value for the owner will be
assigned to each car without an owner. Indeed, our novel query generation algorithms (which also considers keys) is then able to compute the following transformation from $CARS_3$ to $CARS_2$ (expressed as a non-recursive Datalog program with stratified negation):

$$ P_2(p, n, e) \leftarrow P_3(p, n, e) $$
$$ C_2(c, m, p) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) $$
$$ OC_{tmp}(c) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) $$
$$ C_3(c, m, null) \leftarrow C_3(c, m), \neg OC_{tmp}(c) $$

Here, the second rule deals only with cars having an owner; the third rule defines an intermediate relation $OC_{tmp}$ (for Owned Cars) and the fourth rule deals only with cars without an owner (it assigns to them a null value for the owner).

Figure 3 shows a data transformation computed by the queries above. This transformation is indeed more desirable than the one shown in Figure 2, as it avoids the disadvantages shown by current schema mapping and query generation algorithms.

Intuitively, our schema mapping generation algorithm takes specifically care of nullable attributes, and aims at avoiding the generation of useless tuples made of invented and null values only. Furthermore, our query generation algorithm takes specifically into account key constraints, and aims at computing transformations that avoid the generation of multiple tuples in a same relation having the same key; a resolution procedure is adopted to give preference to already existing values rather than to either null values or invented values.

Our second example motivates the need for a mechanism to express more specific value correspondences.

**Example 2.2** Consider the mapping problem shown in Figure 4. The source schema $CARS_3$ is as in Example 2.1. The target schema $CARS_1$ consists of just a single relation, intended to store a tuple for each car, possibly with the name of its owner (or null, otherwise).

For this mapping problem, our mapping algorithms compute the following schema mapping:

$$ C_3(c, m) \rightarrow C_1(c, m, n') $$
$$ P_3(p, n, e) \rightarrow C_1(c', m', n) $$
$$ O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_1(c, m, n) $$

Then, even our mapping algorithms generate the following (uncorrect) transformation:

$$ C_1(c, m, n) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) $$
$$ OC_{tmp} \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) $$
$$ C_1(c, m, null) \leftarrow C_3(c, m), \neg OC_{tmp}(c) $$
$$ C_1(f_C(p, n, e), f_M(p, n, e), n) \leftarrow P_3(p, n, e) $$
Figure 4: Another sample mapping problem

Figure 5 shows a data transformation computed by the queries above. The problem is in the fourth rule, that specifies the generation of a new invented car for each person.

\[
\begin{array}{c|c|c}
\text{P}_3 & \text{C}_3 & \text{O}_3 \\
\hline
p & n & e \\
\hline
\text{p21} & \text{John} & \text{p22} \\
\text{p22} & \text{MJ} & \text{mj0} \\
\end{array}
\begin{array}{c|c|c}
\text{C}_3 & m & c \\
\hline
\text{c85} & \text{Ferrari} & \text{null} \\
\text{c86} & \text{Ford} & \text{null} \\
\gamma_01 & \mu_01 & \text{John} \\
\gamma_02 & \mu_02 & \text{MJ} \\
\end{array}
\begin{array}{c|c|c}
\text{O}_3 & c & p \\
\hline
\text{c85} & \text{p22} \\
\end{array}
\]

Figure 5: A data transformation for the mapping problem of Example 2.2

The problem outlined in the above Example 2.2 is not caused by available mapping algorithms. Rather, the traditional notion of “attribute correspondence” does not permit to express the intended mapping. Indeed, attribute correspondence labeled cn in Figure 4 specifies that each value occurring in attribute name in the source database should occur as well in attribute name in the target database. However, in the desired mapping, we would like that only names of car owners occur in attribute name in the target database. To the best of our knowledge, this kind of correspondences can not be specified in Clio, even resorting to filters [14].

The desired mapping can be expressed by using referenced-attribute correspondences, where the scope of an attribute correspondence can be bounded to values occurring in tuples that can be retrieved by traversing a path of foreign keys.

**Example 2.2 (cont.)** For the same mapping problem of Example 2.2, we would like to let attribute name in the target database correspond to names of car owners, that is, to those values occurring in the source database as name in tuples of relation P_3 that can be retrieved by navigating the foreign key from O_3.person to P_3. This can be specified by using, instead of the traditional attribute correspondence cn, the following referenced-attribute correspondence (symbol \(\rightsquigarrow\) denotes, intuitively, the “traversal” of a foreign key):

\[ cn' : (O_3.person \rightsquigarrow P_3.name, C_1.name) \]
Using these correspondences, our mapping algorithms compute the following schema mapping:

\[ C_3(c, m) \rightarrow C_1(c, m, n') \]
\[ O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_1(c, m, n) \]

Then, our algorithms generate the following correct transformation:

\[ C_1(c, m, n) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \]
\[ OC_{tmp} \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \]
\[ C_1(c, m, \text{null}) \leftarrow C_3(c, m), \neg OC_{tmp}(c) \]

Figure 6 shows a data transformation computed by the queries above; it is the natural and desired transformation.

\[ \begin{array}{c|c|c|c|c}
    P_3 & c & m & n & e \\
    \hline
    \text{p21} & \text{John} & \text{j@...} & \text{p22} & \text{MJ} & \text{mj@...} \\
\end{array} \]
\[ \begin{array}{c|c|c|c|c}
    C_3 & c & m & n & e \\
    \hline
    \text{c85} & \text{Ferrari} & \text{null} & \text{null} \\
    \text{c86} & \text{Ford} & \text{null} & \text{null} \\
\end{array} \]
\[ \begin{array}{c|c|c|c|c}
    O_3 & c & p & e \\
    \hline
    \text{c85} & \text{p22} & \text{null} \\
\end{array} \]

\[ \begin{array}{c|c|c|c|c}
    C_1 & c & m & n & e \\
    \hline
    \text{c85} & \text{Ferrari} & \text{null} & \text{null} \\
\end{array} \]

Figure 6: A better data transformation for the mapping problem of Example 2.2

3 Preliminaries

3.1 Relational Model

A relation schema is a named set of attributes, \( R(A_1, \ldots, A_k) \). A relational schema is a set of relation schemas, \( R = \{ R_1, \ldots, R_n \} \), together with a set \( \Gamma_R \) of integrity constraints (described next). To simplify the presentation, we assume that all attributes are of a same simple type, e.g., strings. At the instance level, a relation is a set of tuples over the attributes of the relation; a relational database is a set of relations.

We consider the following integrity constraints. Attributes of relations can either be nullable or non nullable (i.e., mandatory). By default we assume that attributes are mandatory, and show nullable attributes with a null superscript. Each relation has a primary key (or, simply, a key), comprising one or more non nullable attributes. A key is simple if it consists just of a single attribute; otherwise, it is composite. A key attribute is an attribute that belongs to a key; otherwise, it is a non-key attribute. As it is customary, we will underline key attributes.

A foreign key (or referential constraint) is an attribute of a relation used to reference (the key attribute of) another relation. We show foreign keys by means of dashed arrows. In this paper, we consider foreign keys used to reference simple keys only.

Applicability of our results is related to termination of a (modified) chase procedure, that can be guaranteed, in this framework, by considering relational schemas in which foreign key constraints form a weakly acyclic set [5], something that we will assume in the remainder of the paper.

Specifically, a set of foreign key constraints is weakly acyclic [5] if the dependency graph built has follows has no cycle going through a special arc:
Figure 7: A sample mapping problem

- the dependency graph contains a node for each attribute \( R.A \) of the relational schema;
- the dependency graph contains a directed edge from attribute \( R_1.A_1 \) to attribute \( R_2.A_2 \) for each (simple) foreign key constraints \( R_1.A_1 \unique R_2.A_2 \);
- moreover, the dependency graph contains a special directed edge from attribute \( R_1.A_1 \) to attribute \( R_2.A' \) for each (simple) foreign key constraints \( R_1.A_1 \unique R_2.A_2 \) and attribute \( A' \) in \( R_2 \) different from \( A_2 \).

3.2 Basic Relational Mapping Systems

We now briefly recall basic mapping algorithms. Specifically, as our baseline, we refer here to the schema mapping and query generation algorithms defined in the context of the Clio project, as originally proposed in [14] and then refined in [16, 6], but limited to the flat relational case. Please note that such proposals do consider foreign keys (as inclusion dependencies), nullable attributes (but only in a rather limited way), and do not consider key constraints. To simplify the presentation, we describe here algorithms that do not consider nullable attributes and keys at all. In this section, we refer to the mapping problem depicted in Figure 7.

In a mapping problem, we are given a source schema \( S \), a target schema \( T \), and a set \( C \) of attribute correspondences between these schemas. The goal is deriving a set \( Q \) of queries to compute target database instances from source database instances, comprising a query (i.e., view) \( q_i : S \rightarrow R_i \) for each target relation \( R_i \).

An attribute correspondence (or, simply, correspondence) is a pair of attributes \((R_1.A_1, R_2.A_2)\), where \( A_1 \) is an attribute of a source relation \( R_1 \) and \( A_2 \) is an attribute of a target relation \( R_2 \). When relation names are clear from the context, we will simply write \((A_1, A_2)\). Intuitively, an attribute correspondence \((A_1, A_2)\) specifies a “flow” of data from source attribute \( A_1 \) to target attribute \( A_2 \), that is, the fact that each value occurring in \( A_1 \) in the source database should occur as well in \( A_2 \) in the target database. Hence, it specifies a “value correspondence.” In our visual representation attribute correspondences are shown as solid arrows, directed from a source attribute to a target attribute.

Schema Mapping Generation. The basic schema mapping generation algorithm proceeds in two phases: (i) logical relation generation and (ii) logical mapping generation.

A logical relation, also called tableau, is informally a group of semantically related attributes or tuples in a schema, intended to represents an “autonomous concept” of the schema, such as a car or the ownership of a car by a person. (By “autonomous” we mean that it does not require data in other parts of the schema.) In practice, each logical relation is computed by chasing an
individual relation schema using the constraints defined over its relational schema. The result of logical relation generation is a set of logical relations or tableaux, one set in each schema.

For example, consider the mapping problem shown in Figure 7. The logical relations in the source schema are: (i) \( P_{2a}(p,n,e) \) and (ii) \( C_{2a}(c,m,p), P_{2a}(p,n,e) \). The logical relations in the target schema are: (i) \( P_3(p,n,e) \), (ii) \( C_3(c,m) \), and (iii) \( O_3(c,p), C_3(c,m), P_3(p,n,e) \).

Logical mapping generation has the goal of computing a schema mapping from the source to the target schema. In Clio, a schema mapping \( \Sigma \) is a set of logical mappings (i.e., mapping constraints), where each logical mapping is a source-to-target tuple-generating dependency (s-t tgd) [5], that is, a first-order formula of the following form:

\[
(\forall x)(\phi_S(x) \rightarrow (\exists y)\psi_T(x,y)),
\]

where \( \phi_S \) and \( \psi_T \) are (in Clio) conjunctive queries over the source and target schemas, respectively.

Candidate logical mappings are first computed as follows: A skeleton is a pair of tableaux \((T_1, T_2)\), comprising a source tableau \(T_1\) and a target tableau \(T_2\). A skeleton \((T_1, T_2)\) covers an attribute correspondence \((A_1, A_2)\) if \(A_1\) occurs in \(T_1\) and \(A_2\) occurs in \(T_2\). Let \(V\) be the set of correspondences in \(C\) covered by a skeleton \((T_1, T_2)\); if \(V\) is not empty, the triple \((T_1, T_2, V)\) defines a candidate logical mapping. Not all candidate logical mappings contribute to the schema mapping; rather, subsumed and implied logical mappings should be pruned. (See [14, 16] for details on pruning.)

The remaining candidate logical mappings define the schema mapping, obtained by interpreting each candidate logical mapping \((T_1, T_2, V)\) as a s-t tgd of the form \(\forall T_1 \rightarrow \exists T_2, V\), where \(V\) denotes the conjunction of a set of conditions, comprising a condition \(t_1 = t_2\) for each covered correspondence \((A_1, A_2)\), where \(t_1, t_2\) are the terms occurring in the positions respectively corresponding to attributes \(A_1, A_2\).

For example, the application of the above logical mapping generation algorithm to the mapping problem depicted in Figure 7 leads to the following schema mapping:

\[
P_{2a}(p,n,e) \rightarrow P_3(p,n,e) \\
C_{2a}(c,m,p), P_{2a}(p,n,e) \rightarrow O_3(c,p), C_3(c,m), P_3(p,n,e)
\]

Algorithm 1 summarizes the basic schema mapping generation algorithm [14, 16].

**Algorithm 1 (Basic Schema Mapping Generation)**

**Input:** Source schema \(S\), with constraints \(\Gamma_S\); target schema \(T\), with constraints \(\Gamma_T\); set \(C\) of attribute correspondences.

**Output:** Schema mapping, a set of logical mappings.

1. **Logical Relation Generation.** Compute all source and target logical relations as the tableaux obtained by chasing individual relations in \(S\) and \(T\) with \(\Gamma_S\) and \(\Gamma_T\), respectively.

2. **Identify Candidate Logical Mappings.** For each skeleton \((T_1, T_2)\), compute the set \(V\) of correspondences covered by the skeleton; if \(V\) is non-empty, define a candidate logical mapping \((T_1, T_2, V)\).

3. **Pruning.** Prune those candidate logical mappings that are subsumed by other candidate logical mappings. Then, prune those candidate logical mappings that are implied by other remaining candidate logical mappings.

4. **Actual Schema Mapping Generation.** Generate a logical mapping from each remaining candidate logical mapping.
**Query Generation.** Query generation is concerned with the translation of the schema mapping (a set of logical mappings) into a set of queries, comprising a query for each target relation. In the simplest cases, it is sufficient to “reverse” and unfold the various logical mappings, and interpreting the result as a non-recursive Datalog program [1].

For example, for the mapping problem depicted in Figure 7, we obtain the following transformation:

\[
\begin{align*}
P_3(p, n, e) & \leftarrow P_{2a}(p, n, e) \\
P_3(p, n, e) & \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e) \\
C_3(c, m) & \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e) \\
O_3(c, p) & \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e)
\end{align*}
\]

Figure 8 shows a sample data transformation computed by the queries above.

![Figure 8](image)

Figure 8: A data transformation for the mapping problem of Figure 7

In more complex cases, logical mappings may comprise existentially quantified variables (i.e., variables that occur in the right-hand side of an implication but not in its left-hand side). In this case, existentially quantified variables are skolemized, by replacing each such variable by a different Skolem functor that depends on all the universally quantified variables that occur in the right-hand side of the logical mapping [16]. Because of this, the resulting transformations are, in general, non-recursive Datalog programs with Skolem functors, where Skolem functors are the mechanism used to specify “invented values” that should occur in the target instance [8]. (See, for an example, the transformation of Example 2.1.)

Algorithm 2 summarizes the basic query generation algorithm [14, 16].

**Algorithm 2 (Basic Query Generation)**

**Input:** Source schema \( S \), with constraints \( \Gamma_S \); target schema \( T \), with constraints \( \Gamma_T \); schema mapping \( \Sigma \), a set of logical mappings.

**Output:** A non-recursive skolemized Datalog program, defining a query for each target relation.

1. **Logical Mapping Skolemization and Rewriting.** Skolemize existentially quantified variables in logical mappings in \( \Sigma \) (i.e., those variables that occur in the right-hand side but not in the left-hand side). Rewrite each skolemized logical mapping \( m \) into a set of unitary logical mappings, one for each relational atom in the consequent of \( m \), each having the same premise of \( m \) but, as consequent, just the single relational atom selected in the consequent of \( m \).

2. **Actual Query Generation.** Generate a set of skolemized Datalog rules from the unitary skolemized logical mappings, as follows. From each unitary skolemized logical mapping \( m \), generate a Datalog rule having as body the premise of \( m \) and as head the (single) consequent of \( m \).
4 Referenced-Attribute Correspondences

Recall from [14, 16] that an attribute correspondence is a pair of attributes \((A, B)\) where \(A\) is a source attribute and \(B\) is a target attribute. An attribute correspondence specifies, intuitively, that each value occurring in source attribute \(A\) should occur in target attribute \(B\) as well. Sometimes, this is a too strong specification. Indeed, it is sometimes the case that only a subset of the values occurring in \(A\) should appear in \(B\). For instance, in Example 2.2, \(A\) are names of persons, while \(B\) are names of car owners, where the latter are in general a subset of the former. To the best of our understanding, it is not possible to specify such a correspondence using a traditional value correspondences, even resorting to filters [14]. Rather, in our example, we would like to “filter” values with respect to a join condition involving a foreign key defined in the schema. To this end, we introduce a novel type of value correspondences, called referenced-attribute correspondences, which generalize attribute correspondences.

A referenced attribute is an expression of the following form:

\[ R_1.A_1 \rightsquigarrow \ldots \rightsquigarrow R_n.A_n, \]

where: (i) each \(A_i\) is an attribute of relation \(R_i\), for \(1 \leq i \leq n\), and (ii) in each relation \(R_i\), attribute \(A_i\) references, via a foreign key, the key \(K_{i+1}\) of relation \(R_{i+1}\), for \(1 \leq i < n\). Intuitively, symbol \(\rightsquigarrow\) denotes the traversal of a foreign key. The referenced attribute is the last (i.e., the rightmost) in the path (e.g., \(R_n.A_n\) in the above example). Thus, a referenced attribute is an attribute prefixed by a path of foreign keys. The set of values associated with a referenced attribute comprises only a subset of the values occurring in that attribute, and specifically those values occurring in tuples that can be retrieved by traversing the whole path of foreign keys. For the referenced attribute above, a value \(v\) belongs to its value set (wrt respect to some database instance \(I\)) if there exists a sequence of tuples \(t_1, \ldots, t_n\), respectively in \(R_1, \ldots, R_n\), such that, for \(1 \leq i < n\), \(t_i.A_i\) is equal to the key attribute of \(t_{i+1} \rightsquigarrow v\) is equal to \(t_n.A_n\). For example, in schema \(\text{CARS}_3\) of Example 2.2, referenced attribute \(O_3.\text{person} \rightsquigarrow P_3.\text{name}\) denotes the names of car owners.

A referenced-attribute correspondence (or r-a correspondence) is a pair \((\pi_1, \pi_2)\), where \(\pi_1, \pi_2\) are, respectively, referenced attributes over the source and target schema. For example, \((O_3.\text{person} \rightsquigarrow P_3.\text{name}, C_1.\text{name})\). Intuitively, such r-a correspondence specifies that all values occurring in the source referenced attribute \(\pi_1\) should occur in the target referenced attribute \(\pi_2\) as well.

The management of r-a correspondences in mapping algorithms is as follows. We first need to define notions related to coverage. To this end, a referenced attribute \(R_1.A_1 \rightsquigarrow \ldots \rightsquigarrow R_n.A_n\) is covered by a tableau \(T\) if the following conditions are satisfied: (i) \(T\) contains relation atoms for \(R_1, \ldots, R_n\), and (ii) \(T\) contains equality conditions \(t(R_i.A_i) = t(R_{i+1}.K_{i+1})\), for \(1 \leq i < n\), where \(t(R.A)\) denotes the term occurring in the relation atom for \(R\) in the position for attribute \(A\) and \(K_{i+1}\) denotes the key attribute of relation \(R_{i+1}\). Then, a r-a correspondence \((\pi_1, \pi_2)\) is covered by a skeleton \((T_1, T_2)\) if referenced attributes \(\pi_1, \pi_2\) are respectively covered by tableaux \(T_1, T_2\).

Then, each candidate logical mapping is defined as a triple \((T_1, T_2, V)\), where \(V\) is the (non-empty) set of correspondences covered by skeleton \((T_1, T_2)\). (\(V\) can include both traditional attribute correspondences and referenced-attribute correspondences.) If a candidate logical mapping \((T_1, T_2, V)\) is not pruned, then it contributes to a logical mapping in the schema mapping, a source-to-target tuple generating dependency of the form:

\[ \forall T_1 \rightarrow \exists T_2, V, \]

where \(V\) denotes the conjunction of a set of conditions, comprising, for each covered referenced-attribute correspondence \((R_1.A_1 \rightsquigarrow \ldots \rightsquigarrow R_n.A_n, R'_1.A'_1 \rightsquigarrow \ldots \rightsquigarrow R'_m.A'_m)\), a condition \(t_n = t'_m\).
where \( t_n, t'_m \) are the terms occurring in the positions respectively corresponding to referenced attributes \( A_n, A'_m \).

Of course, as with traditional attribute correspondences, there may be multiple coverages of a same referenced-attribute correspondence by a skeleton. To this end, we need to introduce the following more precise definitions. A coverage mapping of a referenced attribute \( R_1.A_1 \mapsto \ldots \mapsto R_n.A_n \) by a tableau \( T \) is a sequence \( \langle t_1, \ldots, t_n \rangle \) of terms (a term is either a variable or a constant) which occur in \( T \), such that each \( t_i \) occurs in the position for attribute \( R_i.A_i \), for \( 1 \leq i \leq n \), and each \( t_i \) occurs as well in the position for the key attribute \( R_{k+1}.K_{i+1} \) of relation \( R_{k+1} \), for \( 1 \leq i < n \). A coverage mapping of a referenced-attribute correspondence \((\pi_1,\pi_2)\) by a skeleton \((T_1,T_2)\) is a pair of term sequences \( (s_1,s_2) \), where each \( s_i \) is a coverage mapping of referenced attribute \( \pi_i \) in tableau \( T_i \).

A coverage of a skeleton \((T_1,T_2)\) is a set \( V = \{ (v_1,cm_1), \ldots, (v_n,cm_n) \} \) of pairs correspondence-coverage mapping, each covered correspondence \( v_i \) with its coverage mapping \( cm_i \). When coverage mappings are clear from the context (i.e., there can not be ambiguous interpretations), a coverage is simply denoted as a set \( V = \{ v_1, \ldots, v_n \} \) of correspondences.

Then, each candidate logical mapping is a triple of the form \((T_1,T_2,V)\), where \((T_1,T_2)\) is a skeleton and \( V \) is a selected coverage for such skeleton. If a candidate logical mapping \((T_1,T_2,V)\) is not pruned, it then gives rise to a logical mapping in the schema mapping having the following form:

\[
\forall T_1 \rightarrow \exists T_2. V,
\]

where \( V \) denotes the conjunction of a set of conditions, comprising a triple of conditions \( A_n = t_n, A'_m = t'_m, A_n = A'_m \) for each referenced-attribute correspondence \((R_1.A_1 \mapsto \ldots \mapsto R_n.A_n, R'_1.A'_1 \mapsto \ldots \mapsto R'_m.A'_m)\) with coverage mapping \( \langle t_1, \ldots, t_n \rangle, \langle t'_1, \ldots, t'_m \rangle \).

**Example 4.1** Figure 9 shows a mapping problem involving a referenced-attribute correspondence. This is a variant of the problem considered in Example 2.2, in which attribute name in the target schema is mandatory rather than nullable and, of course, a r-a correspondence for names of car owners is used instead of a traditional attribute correspondence.

![Figure 9: A referenced-attribute correspondence](image-url)
This is the best possible schema mapping for this problem. (Note that the second logical mapping is required, since we need to move data about all cars.) Note that the use of a traditional correspondence rather than a r-a correspondence would have led to the generation of the following undesired additional logical mapping, which intuitively specifies that an invented car is needed for each person:

\[ P_3(p, n, e) \rightarrow C_{1a}(c', m', n) \]

From the schema mapping above, our query generation algorithm (described in Section 6) is then able to compute the following transformation, which does not invent cars, and invents names for car owners only for cars without a real owner (remind that owner names are mandatory in the target schema CARS1a):

\[
\begin{align*}
C_{1a}(c, m, n) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \\
OC_{tmp}(c) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \\
C_{1a}(c, m, f_N(c, m)) & \leftarrow C_3(c, m), \neg OC_{tmp}(c)
\end{align*}
\]

A referenced attribute is equivalent to a traditional attribute if its prefix path of foreign keys is empty. Similarly, an r-a correspondence is semantically equivalent to a traditional attribute correspondence if both its referenced attributes are just ordinary attributes. In this sense, r-a correspondences generalize traditional attribute correspondences.

We feel that referenced-attribute correspondences have a natural and intuitive semantics, since they specify correspondences between elements that already exist in the involved schemas (specifically, in each schema, an attribute and a path of foreign keys). Visually, it is possible to specify a r-a correspondence by first drawing an attribute correspondences, and then selecting, in each schema, a path of foreign keys to the relation containing the referenced attribute.

See also Example C.2 in the Appendix for another motivating example for referenced-attribute correspondences.

5 Schema Mapping Generation — with Nullable Attributes

First of all, let us consider the role of null values and invented values in the context of schema mapping. In general, null values and nullable attributes are mechanisms for dealing with incomplete information (see, e.g., [1]). In databases, null values are usually used with the following three main possible semantics: nonexistent (“John hasn’t a car”), unknown (“John has a car, but we don’t know which one”), no-information (“we don’t know if John has a car”). Given the possibility to introduce new invented values in a target instance, we adopt the unknown semantics for invented values (also called labeled nulls), and use the no-information semantics for null values (also called unlabeled nulls). Intuitively, an invented value (to be used in a target instance) denotes a required but unknown value; it is therefore essentially a “placeholder” for a value that should be supplied in the target instance. On the other hand, a null value in a target instance denotes a value that is not available in the source instance and it is not required in the target.

In our framework, we need to consider nullable attributes (mainly) in the context of schema mapping generation, while keys need to be considered (mainly) in the context of query generation. We therefore now extend the basic relational schema mapping generation algorithm outlined in Section 3.2 to take into account nullable attributes. Our management of nullable attributes is based on the following extensions to baseline algorithms: (i) each logical relation can be based on a different combination of null/non-null values for nullable attributes — we define a different notion of tableaux and a modified chase procedure to compute logical relations; (ii) while candidate logical mappings are still based on pairs of logical relations, we extend
logical mapping generation mainly by means of novel pruning rules, to be used together with the subsumption and implication rules, to select the meaningful logical mappings. These topics are analyzed in the following subsections. We assume that the reader is familiar with notions related to the standard chase procedure (see, e.g., [1]).

5.1 Logical Relation Generation

Recall that logical relations are computed separately in each individual schema. In the basic logical relation generation algorithm, each logical relation (or tableau) is computed by applying a standard chase procedure to each individual (base) relation of a schema. Intuitively, each base relation \( R \) gives rise to a logical relation \( T_R \), containing the attributes of \( R \), plus all attributes in relations that can be reached from \( R \) by (recursively) following foreign keys. Each tableau can also be considered as a conjunctive query (called tableau query in [1]). Moreover, if we chase a table query \( T \) using some set \( \Gamma \) of integrity constraints and we obtain another tableau \( T' \), it turns out that \( T \) and \( T' \) are equivalent queries over database satisfying \( \Gamma \). (More precisely, the projection of \( T' \) to the attributes of \( T \) is equivalent to \( T \) over instances satisfying \( \Gamma \).) A further interpretation of a logical relation (or a tableau) is as a “representative instance,” obtained from a tableau by replacing different attributes with distinct constants and interpreting each tableau row as a tuple in a relation. Intuitively, each logical relation describes, by means of a representative instance, a simple autonomous concept represented by the schema; by “autonomous” we mean that it does not require data in other parts of the schema. For instance, the simple autonomous concepts of schema \( CARS_3 \) (shown in Figure 1) introduced in Section 3.2 are represented by the logical relations over that schema:

- \( P_3(p, n, e) \) — persons (unrelated to cars);
- \( C_3(c, m) \) — cars (which in general can be unrelated to persons);
- \( O_3(c, p), C_3(c, m), P_3(p, n, e) \) — owned cars, with their owner.

In presence of nullable attributes, we consider partial tableaux, a variant of the notion of tableaux, and a modified chase procedure, to compute such partial tableaux.

An (ordinary) tableau [1] is a set of relational atoms — of the form \( R(x_1, \ldots, x_n) \), where \( R \) is a relation name and the \( x_i \)'s are variables — together with a set of constraint atoms — of the form \( x_i = x_j \). A partial tableau, apart from relational and constraint atoms as in an ordinary tableau, can also contain null atoms and non-null atoms — respectively of the form \( x_i = null \) and \( x_i \neq null \), where \( x_i \) is a variable bound to some nullable attribute.

Our logical relations are computed, as partial tableaux, each starting from a base relation, by means of a modified chase procedure, as follows. Given a partial tableau \( T \) and a constraint/dependency \( \sigma \), the result of chasing \( T \) with respect to \( \sigma \) is defined by means of the following rules (as usual, we refer to an ordering \( \preceq \) on variables):

**null rule** Let \( \sigma = \text{nullable}(R.A) \) (i.e., attribute \( A \) is nullable in a relation \( R \)), \( R(u) \) be a relational atom in \( T \), and assume that neither \( A = \text{null} \) nor \( A \neq \text{null} \) occur in \( T \). The result of applying \( \sigma \) to \( R(u) \) in \( T \) produces two partial tableaux, \( T' \) and \( T'' \), where \( T' = T \cup \{ A = \text{null} \} \) and \( T'' = T \cup \{ A \neq \text{null} \} \).

**fd rule** Let \( \sigma = R : X \rightarrow A \) be a functional dependency (in particular, a key constraint) over a relation \( R \), and let \( R(u), R(v) \) be two relational atoms in \( T \) such that \( u.X = v.X \) and \( u.A \neq v.A \). Let \( x \) be the least variable in \( \{ u.A, v.A \} \) under the ordering \( \preceq \), and \( y \) be the other one. Call \( \theta \) the substitution that maps \( y \) to \( x \) and is the identity elsewhere. The result of applying \( \sigma \) to \( R(u), R(v) \) in \( T \) is the partial tableau \( \theta(T) \) if \( x \neq y \in T \), and \( \bot \) otherwise.

15
Let $\sigma = R.X \subseteq S.Y$ be an inclusion dependency (in particular, a foreign key constraint), let $R(u)$ be a relational atom in $T$, and suppose that, for each $x \in X$, either attribute $x$ is mandatory in $R$ or $T$ contains a non-null atom $u.x \neq \text{null}$. Moreover, suppose that $T$ does not contain any relational atom $S(v)$ such that $v.Y = u.X$. Let $w$ be a free tuple over $S$ such that $w.Y = u.X$ and $w$ has distinct new variables in all attributes not in $Y$ (that are greater than all variables occurring in $T$). Then, “the” result of applying $\sigma$ to $R(u)$ in $T$ is the partial tableau $T' = T \cup \{S(w)\}$.

Note that there are two main differences with respect to the basic logical relation generation algorithm: (i) a nullable attribute can split a partial tableau into two distinct partial tableaux (one in which the attribute is null, the other one in which the attribute is not null); and (ii) it is possible to traverse a foreign key only if the referencing attribute is mandatory or it is nullable and non-null.

To guarantee termination of the above modified chase procedure, we assume that the integrity constraints over the involved relational schemas comprise only primary key constraints, foreign key constraints (used to reference simple keys and forming a weakly acyclic set [5]), and nullable attribute constraints.

Example 5.1 Consider the target schema CARS$^2$ shown in Figure 2.1. Logical relations for schema CARS$^2$ are the following (partial tableaux can still be considered as “autonomous concepts” or “representative instances” of a schema):

- $P_2(p, n, e)$ — persons (unrelated to cars);
- $C_2(c, m, p), p = \text{null}$ — cars without an owner;
- $C_2(c, m, p), p \neq \text{null}, P_2(p, n, e)$ — owned cars, with their owner.

If we apply our modified chase procedure to a base relation $R$, in general we obtain a set $T_R = \{T_1, \ldots, T_n\}$ of partial tableaux. Note also that each partial tableau can also be considered as a conjunctive query (called tableau query in [1]), possibly with null and non-null conditions. Intuitively, these partial tableaux, seen as queries, form a “partition” of relation $R$, in the following sense: (i) partial tableaux in $T_R$ are partially disjoint, that is, $T_i \cap T_j = \emptyset$ if $i \neq j$ (over databases satisfying the schema constraints); and (ii) $R = T_1 \cup \ldots \cup T_n$. (More precisely, the above conditions hold if we consider each partial tableau $T_i$ as a tableau query restricted to the attributes of $R$, and consider only instances satisfying the integrity constraints of the relational schema of $R$.)

5.2 Logical Mapping Generation

As in basic logical mapping generation, this phase proceeds as follows: (i) skeletons are computed by coupling each source logical relation with each target logical relation; (ii) candidate logical mappings are computed from those skeletons covering at least one correspondence; (iii) some candidate logical mappings are pruned; (iv) the schema mapping is defined from the remaining logical mappings. However, there are a number of differences in our novel algorithm, which we will describe in order.

Computing Candidate Logical Mappings. In the basic framework, an attribute occurring in a tableau is said to be covered by the tableau. With nullable attributes, partial tableaux, and referenced-attribute correspondences, a more complex notion of coverage is needed.

Consider an attribute $A$ of a relation $R$ and a partial tableau $T$. The coverage level of $A$ (or of a variable bound to $A$, thereof) in $T$ can be one of the following:
• **mand**, if \( A \) occurs in \( T \) and it is mandatory in \( R \);
• **null**, if \( A \) occurs in \( T \), it is nullable in \( R \), and \( T \) contains the null condition \( A = \text{null} \);
• **nonnull**, if \( A \) occurs in \( T \), it is nullable in \( R \), and \( T \) contains the non-null condition \( A \neq \text{null} \);
• **none**, if \( A \) does not occur in \( T \).

Then, consider a referenced attribute \( R_1.A_1 \Rightarrow \ldots \Rightarrow R_n.A_n \) and a partial tableau \( T \). The **coverage level of** \( R_1.A_1 \Rightarrow \ldots \Rightarrow R_n.A_n \) **in** \( T \) **is** defined as the coverage level of \( R_n.A_n \) **in** \( T \), provided that all previous attributes \( R_1.A_1, \ldots, R_{n-1}.A_{n-1} \) in the path are covered in \( T \) with level **mand** or **nonnull**; otherwise, the coverage level of the referenced attribute is **none**.

The **coverage degree of a referenced-attribute correspondence** \((\pi_1, \pi_2)\) **by a skeleton** \((T_1, T_2)\) **is a pair** \((c_1, c_2)\), where each \( c_i \) is the coverage level of referenced attribute \( \pi_i \) in tableau \( T_i \).

Of course, the above definitions apply also to ordinary attributes and traditional attribute correspondences.

A skeleton \((T_1, T_2)\) **covers** a referenced-attribute correspondence \((\pi_1, \pi_2)\) (or an attribute correspondence \((A_1, A_2)\)) if the coverage degree of the correspondence is \((c_1, c_2)\), both \( c_1 \) and \( c_2 \) are different from **none**, and \((c_1, c_2)\) is different from \((\text{null, mand})\).²

Of course, the above definitions need to and can easily be extended to consider coverage mappings, as we did in Sections 3.2 and 4.

Let \( V \) be the set of correspondences covered by a skeleton \((T_1, T_2)\). As in basic logical mapping generation, if \( V \) is not empty, the triple \((T_1, T_2, V)\) defines a **candidate logical mapping**. However, not all candidate logical mappings will contribute to the resulting schema mapping: some of them need to be pruned.

**Pruning.** Differently from the basic logical mapping generation algorithm, the pruning phase of candidate logical mappings is based on the following steps: (i) pruning related to nullable attributes; (ii) pruning based on subsumption; (iii) pruning based on implication; and (iv) pruning based on non-null extension. Let us consider the various steps individually.

**Pruning Related to Nullable Attributes.** A candidate logical mapping \((T_1, T_2, V)\) should be pruned if it satisfies one of the following conditions:³

- there is a correspondence in \( V \) having coverage degree \((\text{nonnull, null}), (\text{mand, null}), \) or \((\text{null, nonnull})\) — this rule is motivated by the fact that, if there exists such a candidate logical mapping \( m \), because of the logical relation generation algorithm, there should also be a different but preferable candidate logical mapping \( m' \); for example, \( m \) maps a mandatory attribute to a null nullable attribute, while \( m' \) maps the same mandatory attribute to the same nullable attribute, which has however a non-null value;

- there is a target variable/attribute \( A_2 \) in \( T_2 \) that satisfies the following conditions: (i) attribute \( A_2 \) is nullable and non-null; (ii) there is no foreign key defined from attribute attribute \( A_2 \); and (iii) \( A_2 \) is not bound to any variable/attribute in \( T_1 \) — in this case, we are guaranteed that there is a preferred candidate logical mapping in which a null value is assigned to \( A_2 \) (note that this would not be the preferred semantics if a foreign key starts from attribute \( A_2 \)).

²These conclusions are motivated by a case-by-case reasoning, described in Appendix A.
³These conclusions are motivated by a case-by-case reasoning, described in Appendix A.
Pruning Based on Subsumption and on Implication. As in the basic logical mapping generation, subsumed and implied mappings should be pruned [14, 16].

The basic pruning procedure of the baseline algorithms, based on subsumption and implication, is still valid in our extended algorithm, provided that we re-define the notion of sub-tableau for partial tableaux. A partial tableau \( T' \) is a sub-tableau of a partial tableau \( T \) (written \( T' \subseteq T \)) if:

- the relational atoms of \( T' \) are a superset of the relational atoms in \( T \) (possibly after some renaming of variables);
- the constraint atoms (conditions) in \( T' \) are also a superset of those in \( T \) or they imply them (possibly after renaming of variables);
- the null conditions in \( T' \) are also a superset of those in \( T \) (possibly after renaming of variables); and
- the non-null conditions in \( T' \) are also a superset of those in \( T \) (possibly after renaming of variables).

Moreover, \( T' \) is a strict sub-tableau of \( T \) (written \( T' < T \)) if \( T' \subseteq T \) and the relational atoms in \( T' \) are a strict superset of those in \( T \).

A candidate logical mapping \( m' = (T'_1, T'_2, V') \) is subsumed by another candidate logical mapping \( m = (T_1, T_2, V) \) if \( T'_1 \) and \( T'_2 \) are respective sub-tableaux of \( T_1 \) and \( T_2 \) (with at least one being strict), and \( V = V' \). (That is, if \( T'_1 \subseteq T_1, T'_2 \subseteq T_2, \) and \( (T'_1, T'_2) \) does not cover at least one correspondence that is not already covered by \( (T_1, T_2) \).) Note that, if \( T'_1 \subseteq T_1 \) and \( T'_2 \subseteq T_2 \), then necessarily \( V' \supseteq V \). The subsumption condition says that we should not consider \( m' \) since it covers the same set of correspondences that are covered by the “smaller” (and more general) candidate logical mapping \( m \).

A candidate logical mapping \( m = (T_1, T_2, V) \) is implied by a candidate logical mapping \( m' = (T'_1, T'_2, V') \) whenever \( T_1 = T'_1 \) and \( T_2 = T'_2 \) is a sub-tableau of \( T_2 \) (this implies \( V' \supseteq V \)). Intuitively, all target components (relational atoms, with their additional condition atoms) that are asserted by \( m \) are asserted by \( m' \) as well (with the same conditions).

See [14, 16, 6] for more details on pruning based on subsumption and implication.

Pruning Based on Non-Null Extension. We first introduce the notion of non-null-extension by means of an example. Consider partial tableaux \( T = C_2(c, m, p) \), \( p = \text{null} \) and \( T' = C_2(c, m, p) \), \( p \neq \text{null} \), \( P_2(p, n, e) \) of Example 5.1. Both tableaux have been obtained by chasing the same base relation \( C_2 \), but they have been obtained by posing different conditions over nullable attribute \( p: p = \text{null} \) in \( T \) and \( p \neq \text{null} \) in \( T' \). Moreover, \( T' \) contains a further relational atom, since a foreign key is defined over \( p \). In this case, we say that \( T' \) is a “non-null extension” of \( T \).

Let us now formalize this notion. First, note that each partial tableau can be depicted as a rooted graph (or a rooted tree, if foreign keys are acyclic), whose nodes represent relational atoms, whose arcs represent traversals of foreign keys, and whose root is the base relation from which the chase procedure has been started from. Let \( T \) and \( T' \) be two distinct partial tableaux obtained by chasing a same base relation \( R \) (as described in Section 5.1). We say that \( T' \) is a non-null extension of \( T \) (written \( T' \prec T \)) if \( T' \) can be obtained from \( T' \) by pruning the corresponding rooted graph over one or more nullable foreign keys, with null values for such foreign keys in \( T \) where, for the same attributes, \( T' \) associates non-null values.

For example, for the partial tableaux over schema \( CARS_2 \) of Example 5.1, it turns out that \( C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) \) is a non-null extension of \( C_2(c, m, p), p = \text{null} \).
Non-null extension is similar to, but different from, the sub-tableau relationship. We need further pruning rules, based on non-null extension, which are similar to, but different from, subsumption and implication.

Consider a pair \( m, m' \) of candidate logical mappings satisfying the following conditions: (i) the two mappings are defined over a same source partial tableau; and (ii) the target partial tableaux of \( m' \) is a non-null extension (defined next) of the target partial tableaux of \( m \). In this case, one of the two logical mappings should be pruned, as follows: (a) if the two mappings cover the same correspondences, then \( m' \) should be pruned; or (b) otherwise, if \( m' \) covers more correspondences than \( m \), then \( m \) should be pruned.

We can now formally state our pruning rules based on non-null extension. Let \( m = (T_1, T_2, V) \) and \( m' = (T'_1, T'_2, V') \) be two candidate logical mappings such that: (i) \( T_1 = T'_1 \) and (ii) \( T'_2 \prec T_2 \) (\( T'_2 \) is a non-null extension of \( T_2 \)); then: (a) if \( V = V' \), candidate logical mapping \( m' \) should be pruned, and (b) otherwise, if \( V \subset V' \), candidate logical mapping \( m \) should be pruned.

In case (a), the target partial tableaux of \( m' \) contains more relational atoms than the target partial tableaux of \( m \) but, since the two candidate mappings cover the same correspondences, these additional relational atoms in \( m' \) would contain only terms bound to null values or invented values. Thus, the mapping \( m' \) would produce more tuples in the target instance, but, intuitively, the same “quantity” of information as \( m \); that is, \( m' \) would produce, with respect to \( m \), only additional non-informative tuples, containing just null and invented values. In such case, mapping \( m' \) should be pruned.

In case (b), the target partial tableaux of \( m \) contains at least a foreign key that has been set to null, whereas it will be skolemized in \( m' \) to specify invented values, used to reference additional tuples, at least one of which has an attribute bound to a source attribute. Hence, mapping \( m \) is intended to move less information than \( m' \). In such case, mapping \( m \) should be pruned.

The motivation for cases (a) and (b) are somehow similar to those of subsumption and implication, respectively, but the cases are quite different.

**Actual Schema Mapping Generation.** After pruning, there are a number of remaining candidate logical mappings. If a candidate logical mapping \((T_1, T_2, V)\) has not been pruned, it contributes to a logical mapping in the schema mapping having the following form:

\[
\forall T_1 \rightarrow \exists T_2. V,
\]

where: \( T_1 \) is the source partial tableau (null and non-null conditions included); \( T_2 \) is obtained by the target partial tableau by dropping its null and non-null conditions; and \( V \) denotes the conjunction of a set of conditions, as described in Sections 3.2 and 4.

Of course, the above definitions need to and can easily be extended to consider coverage mappings, as we did in Sections 3.2 and 4.

**Example 5.2** Consider again the mapping problem of Example 2.1, depicted in Figure 1.

We have the following source logical relations:

- \( P_3(p, n, e) \)
- \( C_3(c, m) \)
- \( O_3(c, p), C_3(c, m), P_3(p, n, e) \)

and target logical relations:

\[\text{These conclusions are motivated by a case-by-case reasoning, described in Appendix A.}\]
• $P_2(p, n, e)$
• $C_2(c, m, p), p = \text{null}$
• $C_2(c, m, p), p \neq \text{null}, P_2(p, n, e)$

There are nine skeletons, but only seven different candidate logical mappings, as follows:

• $S_1 : O_3(c, p), C_3(c, m), P_3(p, n, e) / P_2(p, n, e) / P_1, P_2, P_3$
• $S_2 : O_3(c, p), C_3(c, m), P_3(p, n, e) / P_2(p, n, e) / P_1, P_2, P_3$
• $S_3 : C_3(c, m) / C_2(c, m, p), p = \text{null} / c_1, c_2$
• $S_4 : O_3(c, p), C_3(c, m), P_3(p, n, e) / C_2(c, m, p), p = \text{null} / c_1, c_2, o_1$
• $S_5 : C_3(c, m) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / c_1, c_2$
• $S_6 : P_3(p, n, e) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / P_1, P_2, P_3$
• $S_7 : O_3(c, p), C_3(c, m), P_3(p, n, e) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / P_1, P_2, P_3, c_1, c_2, o_1, o_2$

Candidate logical mapping $S_4$ should be excluded (either because of pruning related to nullable attributes, or because of pruning on non-null extension, wrt $S_7$). $S_2$ and $S_6$ are subsumed by $S_1$. There are no implied candidate logical mappings. $S_5$ should be excluded because of pruning on non-null extension (wrt $S_3$).

Therefore, the computed schema mapping is the following:

$$
P_3(p, n, e) \rightarrow P_2(p, n, e)
$$

$$
C_3(c, m) \rightarrow C_2(c, m, p')
$$

$$
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
$$

5.3 Discussion

Algorithm 3 summarizes our schema mapping generation procedure. We have underlined differences with respect to the baseline algorithm described in Section 3.2.

**Algorithm 3 (Schema Mapping Generation)**

**Input:** Source schema $S$, with constraints $\Gamma_S$; target schema $T$, with constraints $\Gamma_T$; set $C$ of referenced-attribute correspondences.

**Output:** Schema mapping, a set of logical mappings.

1. **Logical Relation Generation.** Compute all source and target logical relations as the partial tableaux obtained by chasing, with a modified chase procedure, individual relations in $S$ and $T$ with $\Gamma_S$ and $\Gamma_T$, respectively.

2. **Identify Candidate Logical Mappings.** For each skeleton $(T_1, T_2)$, compute the set $V$ of correspondences covered by the skeleton (use modified notions related to coverage); if $V$ is non-empty, define a candidate logical mapping $(T_1, T_2, V)$.

3. **Pruning.** Perform pruning related to nullable attributes. Prune those candidate logical mappings that are subsumed by other candidate logical mappings. Prune those candidate logical mappings that are implied by other remaining candidate logical mappings. Then, prune the remaining candidate logical mappings according to non-null extensions.
4. **Actual Schema Mapping Generation.** Generate a logical mapping from each remaining candidate logical mapping.

Each schema mapping computed by our mapping generation procedure is a set $\Sigma$ of logical mappings, each a source-to-target tuple-generating dependency [5] of the form:

$$(\forall x)(\phi_S(x) \rightarrow (\exists y)\psi_T(x,y)),$$

where $\phi_S$ is a conjunctive query over the source schema, possibly with null and non-null conditions, and $\psi_T$ is a conjunctive query over the target schema. We call $\phi_S$ and $\psi_T$, respectively, the *premise* and the *consequent* of the logical mapping. Among the integrity constraints defined over the source and target schemas, our algorithm takes into consideration nullable attributes and foreign keys. Keys are ignored by our schema generation procedure; they will be taken into account by our query generation algorithm.

6 Query Generation — with Keys

We now extend the baseline relational query generation algorithm outlined in Section 3.2. The main extension consists in an additional intermediate processing step, concerning the management of key constraints. This step is positioned after logical mapping skolemization and before actual query generation. (As usual, the latter is based on reversing and unfolding of the (modified) logical mappings.)

The novel additional step aims at either ensuring satisfaction of target key constraints or unveiling unsatisfiability of such keys, provided that all integrity constraints over the source schema are satisfied. (When target key constraints are unsatisfiable, we simply signal such inconsistency of the mapping and stop; finding possible repairs of the mapping in such a case is out of the scope of this paper.) This step is based on the following activities: (i) check whether each individual logical mapping is consistent with target key constraints; (ii) identify possible key conflicts between groups of logical mappings, having the same target relation in the consequent of the mapping; (iii) try to resolve the identified key conflicts, by rewriting conflicting logical mappings.

As running example for introducing issues related to query generation, we refer to the mapping problem of Examples 2.1 and 5.2, depicted in Figure 1.

**Logical Mapping Skolemization and Rewriting.** Initially, each logical mapping in $\Sigma$ is a s-t dependency of the form $m_i = \phi_i(x) \rightarrow \psi_i(x, y)$, where $\phi_i$ and $\psi_i$ are conjunctive queries over the source schema $S$ and target schema $T$, respectively ($\phi_i$ may also contain null and non-null conditions). In the mapping above, $x$ is the set of the universally quantified variables of the mapping, which occur in the premise $\phi_i$ of the mapping. These variables are also called the *source variables* of the mapping. Moreover, $y$ are the existentially quantified variables of the mapping. They only occur in the consequent $\psi_i$ of the mapping.

We first skolemize logical mappings, as follows. Let $m_i$ be a logical mapping as above, and $y$ be an existentially quantified variable in $y$, occurring in the consequent of $m_i$. We need to replace each such $y$ either by a null value or by a Skolem functor. If $y$ occurs only in a position for a nullable attribute, we replace $y$ with null. Otherwise, $y$ occurs at least once in a position for a mandatory attribute, and it should be skolemized. In this case, we skolemize $y$, by replacing all occurrences of $y$ with a new Skolem functor term $f_{m_i,y}(w)$, as follows. First, as it is customary in mapping algorithms, we use a different Skolem function $f_{m_i}$ for each different logical mapping $m_i$ and existentially quantified variable $y$. Second, we chose the arguments $w$ of $f_{m_i,y}$ according to the following cases (that are the only possible cases for logical mappings computed by our schema mapping generator):
• $y$ is bound only to a key attribute in $\psi_i$ (specifically, in the relational atom for the “root” relation of the target tableau) — in this case, $w$ consists of all the source variables of the logical mapping $m_i$, that is, $w = x$; equivalently, $w$ can be chosen as any subset of the source variables that include, at least, the variable(s) bound to the key attribute(s) of the “root” relation of the source tableau.

• $y$ is bound only to a non-key attribute in $\psi_i$ — in this case, $w$ consists of the set of term(s) bound to the key attribute(s) of the only relational atom in which $y$ occurs (note that this can lead to nested Skolem terms).

• $y$ is bound in $\psi_i$ to both a non-key foreign-key attribute and a key attribute — in this case, $w$ consists of the set of term(s) bound to the key attribute(s) of the single relational atom in which $y$ occurs as a non-key attribute (this can lead to nested Skolem terms too).

We use a skolemization procedure different from the one adopted by [14, 16] and described in Section 3.2, motivated by the goal of managing target key constraints and the consequent need to possibly obtain functional mappings. See also Appendix B for a discussion about our skolemization procedure.

After this step, skolemized logical mappings are in the form $\phi_i(x) \rightarrow \overline{\psi_i}(x)$, i.e., they do no more contain existentially quantified variables. Moreover, $\overline{\psi_i}$ may now contain Skolem functors and null conditions.

We then rewrite each individual logical mapping $m = \phi_i(x) \rightarrow \overline{\psi_i}(x)$ as a set $\{\phi_i(x) \rightarrow \overline{\psi_{i,j}}(x)\}$ of logical mappings, each having the same premise $\phi_i(x)$ of $m$, but each having a single relational atom in the consequent $\overline{\psi_{i,j}}(x)$, with the related null conditions. (Note that $i$ refers to the original logical mapping $m_i$ in $\Sigma$, and $j$ to a single consequent of $m_i$.) We denote by $\Sigma$ the resulting set of unitary skolemized logical mappings.

**Example 6.1** Consider the logical mappings of Example 5.2. By rewriting them, we obtain the following “unitary” logical mappings:

$$
\begin{align*}
P_3(p, n, e) & \rightarrow_1 P_2(p, n, e) \\
C_3(c, m) & \rightarrow_2 C_2(c, m, null) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow_3 C_2(c, m, p) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow_3 P_2(p, n, e)
\end{align*}
$$

Note how we subscribed each implication arrow, to keep track of the provenance of each unitary mapping (this information will be useful next).

See also Example C.1 in Appendix C for an example involving skolemization.

**Functionality Check.** We say that a unitary logical mapping is *functional* if it can not violate the key constraint of the relational atom in its consequent, provided the source schema constraints are satisfied. For example, the first logical mapping of Example 6.1 is functional, since the fact that $p$ is a key for source relation $P_3$ implies that $p$ is a key also for target relation $P_2$.

The functionality check for a unitary logical mapping $m = \phi_i(x) \rightarrow \overline{\psi_{i,j}}(x)$ is performed as follows. Let $k$ be the variable occurring in the position for the key attribute of the relational symbol for $\overline{\psi_{i,j}}$. For each other non-key attribute position $v$ occurring in $\overline{\psi_{i,j}}(x)$, let $\phi_i^{k,v}(k, v)$ be the projection of $\phi_i(x)$ over $k, v$. Then, $m$ is functional if, for each such variable $v$, it is the case that $\phi_i^{k,v}(k, v) \land \phi_i^{k',v'}(k', v') \land k = k' \land v \neq v'$ is unsatisfiable over instances satisfying the source schema constraints. (Otherwise, if there exists a $v$ such that the above expression is satisfiable, we signal an error and stop.)
A minor modification in the procedure (described next) is needed to consider composite keys and Skolem functors. Let \( k = k_1, \ldots, k_n \) be the set of variables occurring in the positions for the composite key of the relational symbol for \( \overline{\psi}_{i,j} \). Then, the functionality check for a unitary logical mapping \( m = \phi_i(x) \rightarrow \overline{\psi}_{i,j}(x) \) is performed by verifying, for each other non-key attribute position \( v \) occurring in \( \overline{\psi}_{i,j}(x) \), if \( \phi^{k,v}_i(k, v) \land \phi^{k,v}_{i,j}(k', v') \land k = k' \land v \neq v' \) is unsatisfiable over instances satisfying the source schema constraints, where \( \phi^{k,v}_i(k, v) \) denotes the projection of \( \phi_i(x) \) over \( k, v \) and \( k = k' \) denotes the conjunction of conditions use a conjunction of conditions \( k_1 = k'_1, \ldots, k_n = k'_n \).

If the above \( k, k \) and/or \( v \) are not simply variables, but rather terms in \( \overline{\psi}_{i,j}(x) \) composed of Skolem functors, then we need to consider the projection of \( \phi_i(x) \) to all variables occurring in such terms. Consider a condition of the form \( f(x_1, \ldots, x_n) = t \), where \( t \) is another term. Such equality condition is satisfiable only if \( t \) is a function term \( f(x'_1, \ldots, x'_n) \) defined over the same Skolem function \( f \) (thus having, of course, the same arity) and each \( x_i = x'_i \) is satisfiable, for \( 1 \leq i \leq n \). On the other hand, the above equality condition is unsatisfiable if \( t \) is a variable or a null term, or if \( t \) is a functor term based on a different Skolem function.

It turns out that the functionality check can be reduced to an \emph{emptiness} test for a conjunctive query with inequalities, under functional and inclusion dependencies, which, from the form of our logical mappings and of the involved integrity constraints, can be computed by chasing the query with respect to the involved integrity constraints (using our modified chase procedure). Specifically, a query \( q \) is empty if and only if the result of chasing it with the involved integrity constraints is \( \emptyset \) or \( \bot \).

Decidability of the functionality check is a direct consequence of classical results \([10, 11]\). Specifically, computability follows from the fact that each \( \phi_i(x) \) is a conjunctive query, possibly with null and non-null conditions, and the schema constraints is composed of keys, a weakly acyclic set of foreign keys, and nullable attributes.

**Example 6.2** Every unitary logical mapping of Example 6.1 is functional. In particular, functionality of the third mapping is guaranteed by the fact that \( car \) is a key for both source relations \( O_3 \) and \( C_3 \). That mapping would not be functional if a car could have more than one owner.

**Identify Key Conflicts.** Even if each individual logical mapping is functional, it is still possible that the whole set of logical mappings is not functional. In particular, two unitary logical mappings \( m = \phi_i(x) \rightarrow \overline{\psi}_{i,j}(x) \) and \( m' = \phi_{i'}(x) \rightarrow \overline{\psi}_{i'j'}(x) \) are in conflict if a same target relational symbol \( R \) occurs in their consequent and they can generate tuples having a same key, but different values for other attributes.

We check a potential key conflict between \( m \) and \( m' \) by verifying, for each non-key attribute position \( v \) occurring in \( \overline{\psi}_{i,j}(x) \) and different from the key \( k \) of \( R \), if \( \phi^{k,v}_i(k, v) \land \phi^{k,v}_{i,j}(k', v') \land k = k' \land v \neq v' \) is unsatisfiable. If the above expression is satisfiable for some attribute \( v \), then we say that \( m \) and \( m' \) are \emph{key conflicting over} \( v \). As above, a minor modification in the procedure is needed to consider composite keys and Skolem functors. Again, decidability of this check follows from the form of the involved queries and constraints.

Not every key conflict causes a real problem. A key conflict between a group of logical mappings is \emph{hard} if the various mappings are intended to copy distinct source values to the conflicting target attribute. For example, if two mappings can suggest two different owners for a same car. A key conflict is \emph{soft} if at most one of the mappings copies source values to the conflicting target attribute, while other logical mappings move null values and/or invented values.
Example 6.3 Consider again the unitary logical mappings of Example 6.1. The first and fourth mappings generate tuples over a same target relation $P_2$. However, they are not key conflicting. (Indeed, the fourth mapping always generates a subset of the tuples generated by the first mapping.)

On the other hand, the second and third logical mappings are key conflicting over attribute $person$. Indeed, the third logical mapping generates tuples for cars having an owner, with the actual owner in position $person$, while the second logical mapping generates tuples for all cars, always with a null value in position $person$. This key conflict is soft. Note also that these two logical mappings are not key conflicting over attribute $model$. ■

Resolving (Soft) Key Conflicts. When we encounter a hard conflict, we signal an error and stop. On the other hand, soft conflicts can sometimes be resolved, as follows.

We first need a “resolution strategy” for soft key conflicts. In what follows, we make the following (natural) assumptions: (i) copying an existing value from the source to the target is preferable to generating a null value or an invented value; (ii) if it is not possible to copy an existing value from the source to the target in a position for a nullable attribute, a null value is preferable to an invented value; and (iii) different invented values are equally preferable. This is consistent with our position stated at the beginning of Section 5.

Let $\Sigma$ be the set of unitary skolemized logical mappings obtained from logical mapping skolemization and rewriting. Each of them is in the form $m_{i,j} = \phi_i(x) \rightarrow \psi_{i,j}(x)$, where $i$ refers to an original logical mapping $m_i$ in $\Sigma$ and $j$ to a single consequent of $m_i$. Assume, without loss of generality, that the sets of variables of the unitary logical mappings in $\Sigma$ are pairwise disjoint.

For each target relation $R$, the conflicting set $CS_R$ for $R$ is defined as the set of unitary logical mappings having relational symbol $R$ in the consequent. Key conflicts happen only among unitary logical mappings in a same conflicting set.

In what follows, we fix a target relation $R$, the conflicting set $CS_R$ for $R$, and denote by $m = \phi_i(x) \rightarrow \psi_{i,j}(x)$ and $m' = \phi_{i'}(x') \rightarrow \psi_{i',j'}(x')$ two different unitary logical mappings in $CS_R$. Moreover, we write $key(R)$ to denote the key attribute of relation $R$, and $v$ to denote a non-key attribute of $R$.

The key conflict identification step allows us to compute the following relationships between unitary logical mappings:

- $m \sim_v m'$, if there is no key conflict on $v$ between $m$ and $m'$;
- $m \succ_v m'$, if there is a hard key conflict on $v$ between $m$ and $m'$; in this case we signal an error and stop;
- $m \succ_v m'$ (or $m \prec_v m'$), if there is a soft key conflict on $v$ between $m$ and $m'$, and $m$ is preferable to $m'$.

If we need to be more precise, we write $m^c \succ_v^c m'$ to denote the source of the key conflict between $m$ and $m'$ on $v$, where $a, a'$ range over $\{c, n, i\}$, and $c, n, i$ respectively stand for copy, null, and invent, to represent the kind of term occurring in the position for $v$ in the respective mapping. That is, we use $c$ for a term bound to a source variable, $n$ for a variable bound to a null value, and $i$ for a Skolem functor term. The only meaningful relationships are $\succ_v^c$, $\succ_v^n$, and $\succ_v^i$.

The basic key conflict resolution is intuitively as follows. Let $m, m'$ be two mappings that are key conflicting over an attribute; assume that $m'$ is preferable to $m$. In this case, we need to rewrite $m$ by “partially disabling” it when $m'$ is applicable. This disabling is performed by appending to the premise of $m$ the negation of the premise of $m'$, projected to the key attribute of the common target relation of the two mappings. In reality, this basic rewriting is more complex, since there can be more than two unitary logical mappings that conflict.
We now describe the basic key conflict resolution procedure in detail. First, we consider, in turn, each unitary logical mapping $m$. Let $\text{preferableTo}(m)$ the set of the unitary logical mappings in $CS_R$ that are preferable to $m$ for at least a non-key attribute:

$$\text{preferableTo}(m) = \{m' \in CS_R \mid \exists v : m' \triangleright_v m\}.$$ 

If $\text{preferableTo}(m)$ is not empty, we rewrite $m$ by adding to its premise, with a conjunction, $k = k' \land \neg \phi_{\text{key}(R)}(k')$ for each unitary logical mapping $m' \in \text{preferableTo}(m)$, where $\phi_{\text{key}(R)}(k')$ is the projection of the premise of $m'$ on $\text{key}(R)$, and $k$ is the variable bound to $\text{key}(R)$ in $m$. Moreover, we add a similar condition $\hat{k} = k' \land \neg \phi_{\text{key}(R)}(k')$ to the premise of each other unitary logical mapping $\hat{m}$ that has been obtained from the same “original” logical mapping in $\Sigma$ as $m$. ($\hat{k}$ denotes the variable bound to $\text{key}(R)$ in $\hat{m}$.) (At the end, all unitary logical mappings derived from a same original logical mapping will have the same modified premise, modulo renaming of variables.)

**Example 6.4** Consider again the soft key conflict on attribute *person* identified in Example 6.3. First, note that the third mapping is preferable to the second one, since the third mapping copies existing values while the second mapping generates null values in the conflicting position. (Note also that the second mapping is “more applicable” then the third one: the third mapping can be applied only for cars with an owner, while the second mapping can be applied to all cars.) Hence, we rewrite the second mapping to “disable” it when the third mapping can be applied, i.e., only for cars having an owner. By applying the above procedure, we rewrite the two mappings as follows:

$$C_3(c, m), \neg \phi_3(c) \rightarrow_2 C_2(c, m, \text{null})$$
$$O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_3 C_2(c, m, p)$$

where $\phi_3(c)$ is defined as follows:

$$\phi_3(c) = \{c \mid O_3(c, p'), C_3(c, m'), P_3(p', n', e')\}$$

The above basic resolution procedure is able to deal with mappings that are key conflicting among them in a simple way. (By “simple” we mean the cases in which, if we take any two conflicting logical mappings, one of the two is clearly preferable to the other.) However, it is sometimes the case that mappings do key conflict in the following more complex way: mappings are key conflicting over multiple attributes, with different preferences among the mappings with respect to the various involved attributes. For example, $m_1$ may be preferable to $m_2$ for the key conflict on an attribute $a$, but $m_2$ is preferable to $m_1$ for the key conflict on a different attribute $b$. In this case, we rewrite the various key conflicting mappings, and also introduce some new logical mappings, that “fuses” the original mappings, by taking values for $a$ and $b$ respectively from $m_1$ and $m_2$. Intuitively, in the case above, each of the two logical mappings should be disabled when the other mapping is applicable. Then, a new logical mapping should be added, having as premise the conjunction of the premises of the original mappings $m_1$ and $m_2$ (i.e., this new mapping will be applicable when both the original $m_1$ and $m_2$ were applicable), and having in the consequent the common values for the non conflicting attributes, the value from $a$ taken from $m_1$, and the value for $b$ taken from $m_2$. We now describe this key conflict resolution step in detail.

We consider each subset $M \subseteq CS_R$ of (the unrewritten) unitary logical mappings such that each mapping in $M$ is preferable to at least one of the remaining mappings in $M$ over at least an attribute. That is, mappings in $M$ are key conflicting over different attributes in distinct
ways, and there is no a single preferred mapping in $M$. For such a set $M = \{m_1,\ldots,m_n\}$ of mappings, we add to the schema mapping an additional unitary logical mapping $m_M$, built as follows:

- The premise of $m_M$ is the conjunction of the premises of the mappings in $M$, plus a set of conditions $k = k_1 \land \ldots \land k = k_n$, where $k$ is a new variable and each $k_i$, for $1 \leq i \leq n$, is the variable bound to $key(R)$ in mapping $m_i$.

- The consequent of $m_M$ is the relational atom $R(k_t, t_1, \ldots, t_h)$ built as the “fusion” of the consequents of the mappings in $M$, as follows: (i) $k$ is the new variable we introduced in the premise of $m_M$, and it occurs in the consequent in the position for the key attribute of $R$; and (ii) each $t_i$ (in the position for a non-key attribute $v_i$ in $R$) is a term bound to attribute $v_i$ in a mapping $m_{v_i}$ in $M$ for which there is no other mapping $m'$ in $M$ such that $m'$ is preferable to $m_{v_i}$ over $v_i$.

Note that, in the latter item, point (ii), for a position $v_i$ there can be more mappings that are preferable on $v_i$. Any of the preferable terms occurring in position $v_i$ can be used if the various terms are either all bound to source variables or all bound to $null$. However, if the preferable terms are Skolem terms, they can be different Skolem terms; in this case, all such Skolem terms in position $v_i$ should be unified. (Skolem term unification is described next.)

Moreover, let $preferableTo(M)$ the set of unitary logical mappings in $CS_R - M$ that are preferable to at least a mapping $m \in M$ for at least a non-key attribute:

$$preferableTo(M) = \{m' \in CS_R - M \mid \exists v, m \in M : m' \succeq_v m\}.$$ 

If $preferableTo(M)$ is not empty, we rewrite mapping $m_M$ by adding to its premise, with a conjunction, $k = k' \land \neg \phi^{key(R)}_v(k')$ for each unitary logical mapping $m' \in preferableTo(M)$, where $\phi^{key(R)}_v(k')$ is the projection of the premise of $m'$ on $key(R)$, and $k$ is the variable bound to $key(R)$ in $m_M$.

A minor modification in the procedure is needed to deal with composite keys.

**Example 6.5** Let $m_1, m_2$ be two soft key conflicting mappings (let $k$ be the key attribute of $R$), as follows:

$$m_1 : \phi_1(k, a, b) \rightarrow_1 R(k, a, b, c)$$

$$m_2 : \phi_2(k, a, c) \rightarrow_2 R(k, a, b, c)$$

Assume that $m_1, m_2$ are key conflicting over attributes $b$ and $c$, but not over attribute $a$. Furthermore, assume that $m_1$ is preferable to $m_2$ over attribute $b$, but $m_2$ is preferable to $m_1$ over attribute $c$. (E.g., $m_1$ invents new values on $c$, $m_2$ propagates null values on $b$.)

We first rewrite the two mappings (using basic key conflict resolution) as follows. We partially disable each of the mappings when the other is applicable:

$$\phi_1(k, a, b) \land \neg \{\phi^k_2(k)\} \rightarrow_1 R(k, a, b, c)$$

$$\phi_2(k, a, c) \land \neg \{\phi^k_1(k)\} \rightarrow_2 R(k, a, b, c)$$

We then add a new logical mapping that takes into account the case in which the two original mappings $m_1, m_2$ were both applicable; the new mapping picks the best from each individual mapping:

$$\phi_1(k, a, b) \land \phi_2(k', a', c') \land k = k' \rightarrow_{1,2} R(k, a, b, c')$$

26
Example 6.6 Consider the following logical mappings:

\[
m_1 : S_1(k, b), b \neq \text{null}, S_0(k, a) \rightarrow T(k, a, b, c) \\
m_2 : S_2(k, c), S_0(k, a) \rightarrow T(k, a, b, c)
\]

Assume that \(k\) is the key attribute of all relations, \(S_0, S_1, S_2,\) and \(T.\) Moreover, both \(S_1.k\) and \(S_2.k\) references \(S_0.k.\) Attribute \(b\) is nullable in both source and target relations.

The skolemization step produces the following rewriting:

\[
m_1 : S_1(k, b), b \neq \text{null}, S_0(k, a) \rightarrow T(k, a, b, f_c(k, a, b)) \\
m_2 : S_2(k, c), S_0(k, a) \rightarrow T(k, a, \text{null}, c)
\]

Logical mappings \(m_1, m_2\) are key conflicting over attributes \(b\) and \(c,\) but not over attribute \(a.\) Moreover, \(m_1\) is preferable to \(m_2\) over attribute \(b,\) but \(m_2\) is preferable to \(m_1\) over attribute \(c.\) (Indeed, \(m_1\) invents new values on \(c,\) \(m_2\) propagates null values on \(b.\)

We rewrite the two mappings as follows. We first partially disable each of the mappings when the other is applicable:

\[
S_1(k, b), b \neq \text{null}, S_0(k, a) \land \neg\{\phi^k_1(k)\} \rightarrow T(k, a, b, f_c(k, a, b)) \\
S_2(k, c), S_0(k, a) \land \neg\{\phi^k_2(k)\} \rightarrow T(k, a, \text{null}, c)
\]

We then add a new logical mapping that takes into account the case in which the two original mappings \(m_1, m_2\) were both applicable; the new mapping picks the best from each individual mapping:

\[
S_1(k, b), b \neq \text{null}, S_0(k, a), S_2(k', c'), S_0(k', d'), k = k' \rightarrow T(k, a, b, c)
\]

It is sometimes the case that, to resolve a soft key conflict, two (or more) Skolem functors need to be unified, trying to guarantee satisfaction of target key constraints. Let \(f_1(w_1)\) and \(f_2(w_2)\) two functors that need to be unified. Note that, in this case, they should occur in different logical mappings, in a same non key attribute position \(A\) in a target relation \(R.\) (Indeed, if they only occur in a key position, then they do not key conflict. Of course, they can occur also in some key position.) Note also that, by construction, both Skolem functors \(f_1, f_2\) should depend on a same set of terms, consisting of those terms occurring in the positions for the key attributes of relation \(R.\) Of course, the actual terms can be different, but they are in the same number and related to the same attributes. We therefore unify these two functors \(f_1(w_1), f_2(w_2)\) by replacing them with a single functor \(f_{1,2}(w),\) as follows: the Skolem function \(f_{1,2}\) is named from the names \(f_1, f_2\) of the functors to be unified, and its argument \(w\) is obtained from \(w_1, w_2\) by renaming their variables in a common way. The replacement is propagated to all occurrences of these Skolem terms, in the same and in other unitary logical mappings.

Example 6.7 Consider the following logical mappings:

\[
S_1(k, a) \rightarrow T(k, a, b, x) \\
S_2(k, b) \rightarrow T(k, a, b, x)
\]

Assume that \(k\) is the key attribute of \(S_1, S_2,\) and \(T.\) Moreover, assume that all target attributes are mandatory.

The skolemization step produces the following rewriting:

\[
S_1(k, a) \rightarrow T(k, a, f_{b,1}(k), f_{x,1}(k)) \\
S_2(k, b) \rightarrow T(k, f_{a,2}(k), b, f_{x,2}(k))
\]

27
The two mappings are key conflicting on $a$ (first mapping is preferred), on $b$ (second mapping is preferred), and also on $x$ (the two mappings are equally preferable). The key conflict on $x$ can be resolved by unifying Skolem functors $f_{x,1}(k)$ and $f_{x,2}(k)$, rewriting both terms as $f_x(k)$.

The rewritten mapping is the following:

$$S_1(k, a), \neg\{k \mid S_2(k, b')\} \rightarrow T(k, a, f_{b,1}(k), f_x(k))$$

$$S_2(k, b), \neg\{k \mid S_1(k, a')\} \rightarrow T(k, f_{a,2}(k), b, f_x(k))$$

$$S_1(k, a), S_2(k, b) \rightarrow T(k, a, b, f_x(k))$$

**Actual Query Generation.** Our schema mapping consists now of a number of “modified” logical mappings, in which each premise is a conjunctive query with null and non-null conditions, plus a conjunction of safe negations of other conjunctive queries, and each consequent contains just a single relational atom, possibly with null conditions and Skolem terms.

Then, query generation proceeds essentially as in the basic query generation algorithm, by reversing and unfolding each logical mapping. It is also useful to add new rules for defining intermediate relations for the negated subqueries. The result is essentially a non-recursive Datalog program, with Skolem functors and safe negation.

**Example 6.8** Consider again the mapping problem of Examples 2.1, 5.2, 6.1, and 6.4. The current mapping consists of the following logical mappings:

$$P_3(p, n, e) \rightarrow_1 P_2(p, n, e)$$

$$O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_3 P_2(p, n, e)$$

$$C_3(c, m), \neg\{c \mid O_3(c, p'), C_3(c, m'), P_3(p', n', e')\} \rightarrow_2 C_2(c, m, \text{null})$$

$$O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_3 C_2(c, m, p)$$

By “reversing the arrows” and introducing a new intermediate temporary relation when negation is required, we obtain:

$$P_2(p, n, e) \leftarrow P_3(p, n, e)$$

$$P_2(p, n, e) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)$$

$$OC_{tmp}(c) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)$$

$$C_2(c, m, \text{null}) \leftarrow C_3(c, m), \neg OC_{tmp}(c)$$

$$C_2(c, m, p) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)$$

It is then possible to perform some standard query optimization, e.g., the second rule can be dropped, since it is subsumed by the first rule. Figure 3 shows a data transformation computed by the above queries.

6.1 Discussion

Algorithm 4 summarizes our query generation generation procedure. We have underlined differences with respect to the baseline algorithm described in Section 3.2.

We refer the reader to Example C.1 in the Appendix for a more comprehensive example of the application of the query generation algorithm.

**Algorithm 4 (Query Generation)**

28
Input: Source schema $S$, with constraints $\Gamma_S$; target schema $T$, with constraints $\Gamma_T$; schema mapping $\Sigma$, a set of logical mappings.

Output: A non-recursive skolemized Datalog program, with safe negation, defining a query for each target relation.

1. **Logical Mapping Skolemization and Rewriting.** Skolemize existentially quantified variables in the logical mappings in $\Sigma$; use modified skolemization procedure. Rewrite each skolemized logical mapping $m$ into a set of unitary logical mappings, one for each relational atom in the consequent of $m$, each having the same premise of $m$ and, as consequent, just the single relational atom selected in the consequent of $m$.

2. **Functionality Check.** Check whether each unitary skolemized logical mapping is functional. If this is not the case, signal an error and stop.

3. **Manage Key Conflicts.** Identify groups of unitary logical mappings that key conflict. Try to resolve soft key conflicts, by rewriting and/or adding logical mappings. If there are hard or unsolvable key conflicts, signal an error and stop.

4. **Actual Query Generation.** Generate a set of non-recursive skolemized Datalog rules, with a limited form of negation, from the modified unitary skolemized logical mappings, as follows. From each unitary skolemized logical mapping $m$, generate a Datalog rule having as body the premise of $m$ and as head the (single) consequent of $m$.

Our query generation algorithm computes source to target transformations expressed as a non-recursive Datalog program, with Skolem functors (to specify value invention) and safe (stratified) negation.

7 **Related Work**

We use the Clio framework [14, 16] as our reference baseline, as many other mapping algorithms do (e.g., [6, 17]). With respect to these proposals, however, we limit our attention only to the flat relational case, whereas most proposals are focused on nested and XML data. We are aware, of course, that many results in the nested setting can be applied to the flat relational case as well.

Many proposals deal, separately, with the management of different integrity constraints. Most algorithms, starting from Clio [14, 16], are essentially able to deal with foreign keys, in the form of inclusion dependencies. The work in [16] and [19] consider nullable attributes, but not the significant case in which foreign keys can originate from nullable attributes. [19] proposes, in a different context, query resolution to manage target functional dependencies (thus including keys); our key conflict resolution procedure, motivated by a different context, is able to manage a different set of cases.

This paper focuses on a scenario in which the input is a set of value correspondences and the output is a set of query transformations. We are not aware of any other proposal in which keys, foreign keys and nullable constraints are taken together into consideration with respect to the same scenario. This scenario is different from the ones considered by other proposals, in which the input could be a schema mapping (not derived from a set of correspondences) or the output could be a target instance (without the generation of a query transformation). Indeed, some mapping systems deal with issues similar to ours, but at different stages of the mapping process. For example, [9] deals with duplicate elimination (somehow similar to key conflict resolution) during query execution phase. Data exchange algorithms [5, 12] are also able to deal with key constraints, but do not consider an explicit query generation phase.
To the best of our understanding, referenced-attribute correspondences can not be expressed by traditional value correspondences, even resorting to filters [14]; indeed, a filter permits to express a selection based on a condition, involving only attributes occurring in the same relation of the filtered attribute and constants. On the other hand, a referenced-attribute correspondence can “filter” values with respect to more complex conditions, involving join over foreign keys defined in the schema. Referenced-attribute correspondences can be expressed using “structural correspondences,” e.g., builders in Clip [17]. Structural correspondences are indeed an expressive mechanism to specify mappings, but difficult to use, because of their low abstraction level. For instance, a builder can be used to specify any arbitrary Cartesian product, i.e., something that is not initially present in a schema. Viceversa, r-a correspondences, even if less expressive than builders, have the advantage of referring only to elements already present in a schema.

Therefore, our proposal is unique in focusing on schema mapping and query generation algorithms for relational databases with keys, foreign keys, and nullable attributes, when mappings are initially specified as value correspondences.

8 Conclusions

In this paper, we extended the original schema mapping and query generation algorithms proposed by Clio [14, 16] to deal with keys, foreign keys and nullable attributes, in a comprehensive way, in relational mappings. Specifically, the novelty of our approach consists of: (i) an explicit and broader management of nullable attributes, including the case in which a foreign key is defined over a nullable attribute; (ii) an explicit management of (target) key constraints during query generation. We also introduced referenced-attribute correspondences, which allow to express, as value correspondences, more precise mappings than traditional attribute correspondences.

The algorithms presented in this paper have been implemented and tested over a rich set of cases.

In our future work, we aim to apply and extend algorithms presented in this paper to an object-relational setting [15]. In this case, keys, foreign keys and nullable attributes play a significant role, and flat structures are adequate. Starting from an object-relational mapping visually specified as a set of correspondences/lines, we would like to generate an executable mapping as a set of bidirectional views (query views and update views), as in [13].

Our algorithms can also be extended to mappings over nested/XML data.

Several authors (see, e.g., [19, 5]) have argued that a natural semantics for schema mappings is that based on canonical (universal) solutions/instances. Given a schema mapping (i.e., a set of source-to-target dependencies Σ) and a source instance, a canonical (universal) solution can be intuitively constructed by chasing the source instance with both the dependencies in Σ and the integrity constraints defined over the target schema. Our transformations have a semantics closer to this canonical semantics than transformations computed by basic mapping algorithms. (See, e.g., the motivating examples in the Introduction.) In our future work, we also aim at determining whether our generation algorithms compute canonical/universal target instances, or how they should be modified to obtain such semantics.

Acknowledgements

I would like to thank Gabriele Rendina, who implemented most of the algorithms presented in this paper, and Paolo Papotti, for many useful discussions on schema mapping and related topics.
References


A Notes on the Management of Nullable Attributes

Our management of nullable attributes is mainly based on the following aspects: (i) partial tableaux with null and non-null conditions; (ii) coverage degree for correspondences and attributes; (iii) pruning related to nullable attributes; and (iv) pruning based on non-null extension. We now discuss, mainly by means of examples, our choices related to the management of nullable attributes with respect to points (ii)-(iv) above.

Let us first review cases not involving nullable attributes.

Example A.1 Source/target schemas:

\[ P_s(\text{person}, \text{name}, \text{email}) \quad P_t(\text{person}, \text{name}, \text{email}) \]

Correspondences: on person, name, and email.

Desired mapping:

\[ P_s(p, n, e) \rightarrow P_t(p, n, e) \]

In this case, the coverage degree for the correspondence on email is mand-mand. This is the simplest case managed by basic algorithms.

Example A.2 Source/target schemas:

\[ P_s(\text{person}, \text{name}, \text{email}) \quad P_t(\text{pid}, \text{name}, \text{email}) \]

Correspondences: on name and email. But not on person/pid.

Desired mapping:

\[ P_s(p, n, e) \rightarrow P_t(p', n, e) \]

In this case, the coverage degree for the correspondence on email is mand-mand.

Note however that attributes person and pid and not involved in any correspondence; it is therefore not possible to define the coverage degree for an undefined correspondence. However, we say that the coverage degree for attribute person is mand-none, and the coverage degree for attribute pid is none-mand.

Basic mapping algorithms manage this case by Skolemizing variable \( p' \) and ignoring variable \( p \).

Example A.3 Source/target schemas:

\[ P_s(\text{person}, \text{name}) \quad P_t(\text{person}, \text{name}, \text{email}) \]

Correspondences: on person and name.

Desired mapping:

\[ P_s(p, n) \rightarrow P_t(p, n, e') \]

The coverage degree for attribute email is none-mand.

Basic mapping algorithms manage this case by Skolemizing variable \( e' \).
Let us now consider cases involving nullable attributes.

**Example A.4** Source/target schemas:

$P_s(person, name)$  
$P_t(person, name, email^{null})$

Correspondences: on $person$ and $name$.

Desired mapping:

$P_s(p, n) \rightarrow P_t(p, n, e')$

Desired transformation:

$P_t(p, n, null) \leftarrow P_s(p, n)$

Intuitively, assigning a null value is the best policy for a target attribute $A$ satisfying the following conditions: (i) $A$ is a target attribute, not bound to any source attribute, (ii) $A$ is not an attribute referencing a foreign key. (A worst solution, in this case, consists in skolemizing the corresponding variable.)

There are two candidate logical mappings:

1. $P_s(p, n)(p, n) / P_t(p, n, e'), e' = null / p, n$
2. $P_s(p, n)(p, n) / P_t(p, n, e'), e' \neq null / p, n$

Both logical mappings cover the same correspondences. In the first mapping, the coverage degree for attribute $email$ is none-null. In the second mapping, it is none-nonnull. In this case, we prefer the first mapping, and discard the second one.

**Example A.5** Source/target schemas:

$P_s(person, name, email)$  
$P_t(person, data ⇝ PD_t^{null})$

$PD_t(data, name, email)$

Correspondences: on $person$, $name$, and $email$.

Desired mapping:

$P_s(p, n, e) \rightarrow P_t(p, d'), PD_t(d', n, e)$

Desired transformation:

$P_t(p, f_d(p)) \leftarrow P_s(p, n, e)$

$PD_t(f_d(p), n, e) \leftarrow P_s(p, n, e)$

Candidate logical mappings:

1. $P_s(p, n, e) / P_t(p, d'), d' = null / p$
2. $P_s(p, n, e) / P_t(p, d'), d' \neq null, PD_t(d', n, e) / p, n, e$

Note that, in the second logical mapping, the coverage degree for attribute $data$ is none-nonnull. However, we do not want to discard this mapping (as in Example A.4), because a foreign key starts from attribute $data$.

Note also that, in the first logical mapping, the coverage degree for attribute $data$ is none-null.

We discard the first logical mapping, with respect to the second one, because of pruning based on non-null extension, that is, because: (i) the two mappings are defined over the same source tableau, (ii) the partial tableau $P_t(p, d'), d' \neq null, PD_t(d', e')$ (in the second logical mapping) is a non-null extension of partial tableau $P_t(p, d'), d' = null$ (in the first logical mapping), and (iii) the second logical mapping covers more correspondences than the first one.
Example A.6  Source/target schemas:

\[
\begin{align*}
P_s(person, name) & \quad P_t(person, data) \leadsto PD_t^{null} \\
PD_t(data, email) &
\end{align*}
\]

Correspondences: on person.
Desired mapping:

\[
P_s(p, n) \quad \Rightarrow \quad P_t(p, d')
\]

Desired transformation:

\[
P_t(p, null) \quad \leftarrow \quad P_s(p, n)
\]

Candidate logical mappings:

• \( P_s(p, n) / P_t(p, d'), d' = null / p \)
• \( P_s(p, n) / P_t(p, d'), d' \neq null, PD_t(d', e') / p \)

This example is similar to Example A.5. However, in this case, the second mapping is useless, since it specifies a mapping generating in \( PD_t \) tuples made only of invented value. We discard this second logical mapping, with respect to the first one, because of pruning based on non-null extension, that is, because: (i) the two mappings are defined over the same source tableau, (ii) the partial tableau \( P_t(p, d'), d' \neq null, PD_t(d', e') \) (in the second logical mapping) is a non-null extension of partial tableau \( P_t(p, d'), d' = null \) (in the first logical mapping), and (iii) the second logical mapping does not cover more correspondences than the first one.

To summarize, up to this point, we derive from Examples A.4, A.5, and A.6 the following rules for managing an attribute \( A \) that is nullable in the target schema and is not bound to any variable in the source schema: (i) if \( A \) does not reference a foreign key (Example A.4), discard those logical mappings whose target partial tableau contains condition \( A \neq null \); (ii) otherwise, if \( A \) references a foreign key (Examples A.5 and A.6), consider pruning based on non-null extension.

Example A.7  Source/target schemas:

\[
\begin{align*}
P_s(person, name, email^{null}) & \quad P_t(person, name, email)
\end{align*}
\]

Correspondences: on person, name, and email.
Desired mapping:

\[
\begin{align*}
P_s(p, n, e), e \neq null & \quad \Rightarrow \quad P_t(p, n, e) \\
P_s(p, n, e), e = null & \quad \Rightarrow \quad P_t(p, n, e')
\end{align*}
\]

Desired transformation:

\[
\begin{align*}
P_t(p, n, e) & \quad \leftarrow \quad P_s(p, n, e), e \neq null \\
P_t(p, n, f_e(p)) & \quad \leftarrow \quad P_s(p, n, null), \neg\{ p \mid P_s(p', n', e'), e' \neq null \}
\end{align*}
\]

Intuitively, since attribute email is mandatory in the target schema, new invented values can be used to replace null values.

Candidate logical mappings:
• $P_s(p, n, e), e \neq \text{null} / P_t(p, n, e) / p, n, e$
• $P_s(p, n, e), e = \text{null} / P_t(p, n, e') / p, n$

In the first mapping, the coverage degree for the correspondence on email is nonnull-\text{nonnull} (i.e., the correspondence is considered to be covered). On the other hand, in the second mapping, the coverage degree for attribute email is null-\text{nonnull}, and the correspondence on email is not covered.

Example A.8 Source/target schemas:

\[ P_s(\text{person}, \text{name}, \text{email}) \quad P_t(\text{person}, \text{name}, \text{email}^{\text{null}}) \]

Correspondences: on person, name, and email.

Desired mapping:

\[ P_s(p, n, e) \rightarrow P_t(p, n) \]

Nulls values need not to be propagated, in this case.

Candidate logical mappings:

• $P_s(p, n, e) / P_t(p, n, e'), e' = \text{null} / p, n$
• $P_s(p, n, e) / P_t(p, n, e), e \neq \text{null} / p, n, e$

The first logical mapping should be pruned. In the first mapping, the coverage degree for the correspondence on email is \text{nonnull-null}; the correspondence is not covered.

Example A.9 Source/target schemas:

\[ P_s(\text{person}, \text{name}, \text{email}^{\text{null}}) \quad P_t(\text{person}, \text{name}, \text{email}^{\text{null}}) \]

Correspondences: on person, name, and email.

Desired mapping:

\[ P_s(p, n, e), e \neq \text{null} \rightarrow P_t(p, n, e) \]
\[ P_s(p, n, e), e = \text{null} \rightarrow P_t(p, n, e') \]

Desired transformation:

\[ P_t(p, n, e) \leftarrow P_s(p, n, e), e \neq \text{null} \]
\[ P_t(p, n, \text{null}) \leftarrow P_s(p, n, e), e = \text{null} \]

Candidate logical mappings:

• $P_s(p, n, e), e = \text{null} / P_t(p, n, e'), e' = \text{null} / p, n$
• $P_s(p, n, e), e = \text{null} / P_t(p, n, e'), e' \neq \text{null} / p, n$
• $P_s(p, n, e), e \neq \text{null} / P_t(p, n, e'), e' = \text{null} / p, n$
• $P_s(p, n, e), e \neq \text{null} / P_t(p, n, e), e \neq \text{null} / p, n, e$

The second and third logical mappings should be pruned. For these two logical mappings, coverage degree for the correspondence on email is, respectively, null-nonnull and nonnull-null. We should also keep the first and fourth logical mappings. For these two logical mappings, coverage degree for the correspondence on email is, respectively, null-null and nonnull-nonnull.
Example A.10 Source/target schemas:

\[ P_s(\text{person, name, email}^{null}) \quad P_t(\text{person, name}) \]

Correspondences: on person and name.
Desired mapping:

\[
P_s(p, n, e), e \neq \text{null} \rightarrow P_t(p, n) \]
\[
P_s(p, n, e), e = \text{null} \rightarrow P_t(p, n)
\]

Candidate logical mappings:

- \( P_s(p, n, e), e = \text{null} / P_t(p, n) / p, n \)
- \( P_s(p, n, e), e \neq \text{null} / P_t(p, n) / p, n \)

We should keep both logical mappings. For them, coverage degree for attribute email is, respectively, null-none and nonnull-none.

B Notes on Skolemization

Motivated by the goal of managing target key constraints and the need to possibly obtain functional mappings, we use a skolemization procedure (described in Section 6) different from the one adopted by [14, 16], where each Skolem functor depends on all the universally quantified variables occurring in the premise of the mapping.

It is worth noting that different skolemization procedures in schema mapping algorithms are possible and have been proposed in the literature. As [6], we believe that there is no a single “correct” way of skolemizing a schema mapping; rather, mapping systems should suggest a default skolemization (known to work in most cases), but also allow their (advanced) users to customize Skolem functions, in order to express the desired mappings.

Here we propose a brief comparison of skolemization procedures in mapping algorithms.

Skolem functors can used in logic programs and queries to manage value invention [8]. A query or a program is first evaluated to generate a set of facts possibly containing ground Skolem functors, and then each different ground Skolem functor is replaced by a different new invented value. The way we choose Skolem functions and their arguments affects, among other things, the number of invented values and the quantity of tuples in the result of queries and programs.

Let us consider a logical mapping that need to be skolemized. It is, in general, a s-t dependency of the following form:

\[ m_i : \phi_i(x) \rightarrow \psi_i(x, y) \]

where \( \phi_i \) and \( \psi_i \) are conjunctive queries over the source schema S and target schema T, respectively (\( \phi_i \) may also contain null and non-null conditions). Variables in \( x \) are existentially quantified. They are called source variables. On the other hand, variables in \( y \) are existentially quantified; we will skolemize them. Note that the set \( x \) of source variables can be partitioned into two sets, \( x_{pc} \) and \( x_{po} \), where \( x_{pc} \) (premise-and-consequent) are those variables that also occur in the consequent \( \psi_i \) of the logical mapping, while \( x_{po} \) (premise-only) are those variables that do not occur in \( \psi_i \).

A skolemization procedure replaces each variable \( y \in y \) with a Skolem term. It is generally agreed that, in mapping algorithms, a different Skolem function is used for each different logical mapping and variable. According to this, variable \( y \) in mapping \( m_i \) will be replaced by a Skolem term based on a Skolem function called \( f_{i,y} \). However, the various skolemization procedures proposed in literature differ on the arguments of such Skolem function.
• An extreme case is the one proposed by [2], in which functor \( f_{i,y} \) depends on \( x \), the whole set of source variables occurring in the logical mapping. Let us call \textsc{All-Source-Vars} this skolemization procedure. \textsc{All-Source-Vars} generates the largest set of invented values and target facts. In the Clio setting (with no nullable attributes and no target key constraints), \textsc{All-Source-Vars} computes the canonical solution, which is a universal solution. (See [5] for definitions of ‘solution’, ‘canonical solution’, and ‘universal solution’.)

• On the other hand, [16] proposes that \( f_{i,y} \) depends only on \( x_{pc} \), the subset of the source variables that also occur in the consequent of the logical mapping. Let us call \textsc{Source-and-RHS-Vars} this skolemization procedure. \textsc{Source-and-RHS-Vars} can generate less invented values and compute less target facts than \textsc{All-Source-Vars}. In the Clio setting, \textsc{Source-and-RHS-Vars} does not always compute the canonical solution; sometimes, it does not even compute an universal solution.

• We propose that \( f_{i,y} \) depends on \( x \) if \( y \) is only bound to a key attribute in the consequent of the mapping, but to a subset of \( x \) if it is also bound to some non-key attribute. (See Section 6 for a more precise description of our skolemization procedure.) Let us call \textsc{All-Source-Or-Key-Vars} our skolemization procedure. \textsc{All-Source-Or-Key-Vars} can generate less invented values and compute less target facts than \textsc{All-Source-Vars}. It is also not comparable with \textsc{Source-and-RHS-Vars}. In the Clio setting, \textsc{All-Source-Or-Key-Vars} does not always compute the canonical solution; sometimes, it does not even compute an universal solution.

• Moreover, we considered another different skolemization procedure, as follows. If variable \( y \) occurs only in a position for a nullable attribute (in the target schema), we replace \( y \) with \textit{null}. Otherwise, \( y \) occurs at least once in a position for a mandatory attribute, and we need to skolemize \( y \). Let \( R(u) \) be a relational atom in the consequent of \( m \) that is either (i) the only atom containing \( y \) in a non-key position (if \( y \) does never occur as a key) or (ii) an atom containing variable \( y \) in a position for a key attribute. Moreover, let \( \text{vars}(R(u)) \) be the set of source (i.e., universally quantified) variables occurring in \( R(u) \) or in any other relational atom in the consequent \( \psi_i \) of \( m_i \) whose key is, directly or indirectly, referenced by variables in \( R(u) \). We then replace all occurrences of \( y \) by a new Skolem functor \( f_{i,y}(\text{vars}(R(u))) \).

Call \textsc{Source-Here-and-Ref-Vars} this skolemization procedure. Note that each Skolem function depends at most on \( x_{pc} \), the subset of the source variables that also occur in the consequent of the logical mapping. Therefore, \textsc{Source-Here-and-Ref-Vars} can generate less invented values and compute less target facts than \textsc{Source-and-RHS-Vars}. It is also not comparable with \textsc{All-Source-Or-Key-Vars}. In the Clio setting, \textsc{Source-Here-and-Ref-Vars} does not always compute the canonical solution; sometimes, it does not even compute an universal solution.

Let us first consider a number of cases in which the schema mapping needs to be skolemized. (Variables in key-attribute positions are underlined. Variable to be skolemized are starred.)

Example B.1 Logical schema mapping:

\[ \text{Student}_s(id, name, school) \rightarrow \text{Student}_t(*key, name, school) \]

Possible skolemizations:

• \textsc{All-Source-Vars} — \( f_{key}(id, name, school) \) — a tuple in \( \text{Student}_t \) for each tuple in \( \text{Student}_s \)
• **SOURCE-AND-RHS-VARS** — $f_{key}(name, school)$ — tuples in $Student_s$ with the same values on $name$ and $school$ are collapsed in $Student_t$

• **ALL-SOURCE-OR-KEY-VARS** — either $f_{key}(id, name, school)$ or $f_{key}(id)$ — same effect as **ALL-SOURCE-VARS**

• **SOURCE-HERE-AND-REF-VARS** — $f_{key}(name, school)$ — same as **SOURCE-AND-RHS-VARS**

Source instance:

- $Student_s(\alpha, john, math), Student_s(\beta, john, math), Student_s(\gamma, mary, math), Student_s(\delta, mary, cs)$

Target instances:

- **ALL-SOURCE-VARS** and **ALL-SOURCE-OR-KEY-VARS** — $Student_s(\alpha, john, math), Student_s(\beta, john, math), Student_s(\gamma, mary, math), Student_s(\delta, mary, cs)$

• **SOURCE-AND-RHS-VARS** and **SOURCE-HERE-AND-REF-VARS** — $Student_s(\alpha\beta, john, math), Student_s(\gamma\delta, mary, math), Student_s(\delta, mary, cs)$

Both solutions are universal solutions.

---

**Example B.2** Logical schema mapping:

$$Student_s(id, name, school) \rightarrow Student_t(*key, name, *email)$$

Possible skolemizations:

• **ALL-SOURCE-VARS** — $f_{key}(id, name, school), f_{email}(id, name, school)$ — a tuple in $Student_t$ for each tuple in $Student_s$ — a different $email$ for each different source student

• **SOURCE-AND-RHS-VARS** — $f_{key}(name), f_{email}(name)$ — tuples in $Student_s$ with the same values on $name$ are collapsed in $Student_t$ — a different $email$ for each different target student $name$

• **ALL-SOURCE-OR-KEY-VARS** — $f_{key}(id, name, school), f_{email}(f_{key}(id, name, school))$ — a tuple in $Student_t$ for each tuple in $Student_s$ — a different $email$ for each different target student — skolemization is different from, but the effect is the same as **ALL-SOURCE-VARS**

• **SOURCE-HERE-AND-REF-VARS** — $f_{key}(name), f_{email}(name)$ — same as **SOURCE-AND-RHS-VARS**

Source instance:

- $Student_s(\alpha, john, math), Student_s(\beta, john, math), Student_s(\gamma, mary, math), Student_s(\delta, mary, cs)$

Target instances:

- **ALL-SOURCE-VARS** and **ALL-SOURCE-OR-KEY-VARS** — $Student_s(\alpha, john, email_\alpha), Student_s(\beta, john, email_\beta), Student_s(\gamma, mary, email_\gamma), Student_s(\delta, mary, email_\delta)$

• **SOURCE-AND-RHS-VARS** and **SOURCE-HERE-AND-REF-VARS** — $Student_s(\alpha\beta, john, email_\alpha\beta), Student_s(\gamma\delta, mary, email_\gamma\delta)$

Both solutions are universal solutions.

---

38
Example B.3 Logical schema mapping:

\[ \text{Student}_s(id, \text{name}, \text{schoolname}) \rightarrow \text{Student}_t(id, \text{name}, \ast \text{sid}), \text{School}_t(\ast \text{sid}, \text{schoolname}) \]

Possible skolemizations:

- **ALL-SOURCE-VARS** — \( f_{\text{sid}}(id, \text{name}, \text{schoolname}) \) — a tuple in \( \text{School}_t \) for each tuple in \( \text{Student}_s \) — mapping is functional

- **SOURCE-AND-RHS-VARS** — \( f_{\text{sid}}(\text{name}, \text{schoolname}) \) — a tuple in \( \text{School}_t \) for each different pair \( \text{name}, \text{schoolname} \) — mapping is functional

- **ALL-SOURCE-OR-KEY-VARS** — \( f_{\text{sid}}(id) \) — skolemization is different from, but the effect is the same as **ALL-SOURCE-VARS**

- **SOURCE-HERE-AND-REF-VARS** — \( f_{\text{sid}}(\text{schoolname}) \) — a tuple in \( \text{School}_t \) for each different \( \text{schoolname} \) — mapping is functional

Source instance:

- \( \text{Student}_s(a, \text{john}, \text{math}), \text{Student}_s(b, \text{john}, \text{math}), \text{Student}_s(c, \text{mary}, \text{math}), \text{Student}_s(d, \text{mary}, \text{cs}) \)

Target instances:

- **ALL-SOURCE-VARS and ALL-SOURCE-OR-KEY-VARS** — \( \text{Student}_t(a, \text{john}, \alpha), \text{School}_t(\alpha, \text{math}), \text{Student}_t(b, \text{john}, \beta), \text{School}_t(\beta, \text{math}), \text{Student}_t(c, \text{mary}, \gamma), \text{School}_t(\gamma, \text{math}), \text{Student}_t(d, \text{mary}, \delta), \text{School}_t(\delta, \text{cs}) \)

- **SOURCE-AND-RHS-VARS** — \( \text{Student}_t(a, \text{john}, \alpha\beta), \text{School}_t(\alpha\beta, \text{math}), \text{Student}_t(b, \text{john}, \alpha\beta), \text{Student}_t(c, \text{mary}, \gamma), \text{School}_t(\gamma, \text{math}), \text{Student}_t(d, \text{mary}, \delta), \text{School}_t(\delta, \text{cs}) \)

- **SOURCE-HERE-AND-REF-VARS** — \( \text{Student}_t(a, \text{john}, \sigma_m), \text{School}_t(\sigma_m, \text{math}), \text{Student}_t(b, \text{john}, \sigma_m), \text{Student}_t(c, \text{mary}, \sigma_m), \text{Student}_t(d, \text{mary}, \sigma_{cs}), \text{School}_t(\sigma_{cs}, \text{cs}) \)

Note that, among these solutions, only the former is a universal solution!

Example B.4 Logical schema mapping:

\[ \text{Student}_s(id, \text{name}, \text{sid}), \text{School}_s(\text{sid}, \text{scname}) \rightarrow \text{Student}_t(id, \text{name}, \text{sid}), \text{School}_t(\text{sid}, \text{scname}, \ast \text{city}) \]

Possible skolemizations:

- **ALL-SOURCE-VARS** — \( f_{\text{city}}(id, \text{name}, \text{sid}, \text{scname}) \) — a tuple in \( \text{School}_t \) for each tuple in \( \text{Student}_s \) — mapping is not functional

- **SOURCE-AND-RHS-VARS** — \( f_{\text{city}}(id, \text{name}, \text{sid}, \text{scname}) \) — a tuple in \( \text{School}_t \) for each tuple in \( \text{Student}_s \) — mapping is not functional

- **ALL-SOURCE-OR-KEY-VARS** — \( f_{\text{city}}(\text{sid}) \) — a tuple in \( \text{School}_t \) for each tuple in \( \text{School}_s \) — mapping is functional

- **SOURCE-HERE-AND-REF-VARS** — \( f_{\text{city}}(\text{sid}, \text{scname}) \) — a tuple in \( \text{School}_t \) for each different \( \text{sid} \) — mapping is functional

Source instance:

- \( \text{School}_s(m, \text{math}), \text{School}_s(c, \text{cs}), \text{Student}_s(a, \text{john}, m), \text{Student}_s(b, \text{john}, m), \text{Student}_s(c, \text{mary}, m), \text{Student}_s(d, \text{mary}, c) \)
Both solutions are universal solutions. However, to obtain a “correct” solution, the first solution needs to be chased with respect to target key constraints. On the other hand, the second solution is already “correct.”

Example B.5 Logical schema mapping:

$$Student_s(id, name, schoolname) \rightarrow School_t(*sid, schoolname)$$

Possible skolemizations:

- **All-Source-Vars and All-Source-Or-Key-Vars** — $$f_{sid}(id, name, schoolname)$$ — a tuple in $$School_t$$ for each tuple in $$Student_s$$ — mapping is functional
- **Source-and-RHS-Vars and Source-Here-and-Ref-Vars** — $$f_{sid}(schoolname)$$ — a tuple in $$School_t$$ for each different school name — mapping is functional

Source instance:

- $$Student_s(a, john, math), Student_s(b, john, math), Student_s(c, mary, math), Student_s(d, mary, cs)$$

Target instances:

- **All-Source-Vars and All-Source-Or-Key-Vars** — $$School_s(\gamma_1, math), School_s(\gamma_2, math), School_s(\gamma_3, math), School_s(\gamma_4, cs)$$
- **Source-and-RHS-Vars and Source-Here-and-Ref-Vars** — $$School_s(\gamma_n, math), School_s(\gamma_n, cs)$$

Both solutions are universal solutions.

According to the above examples, it turns out that, among the proposed skolemization procedures, only All-Source-Or-Key-Vars always yields universal and functional solutions. Another possible criteria is the size of the computed result. Usually All-Source-Or-Key-Vars provides a large result set (as large as All-Source-Vars), unless the result computed by All-Source-Vars is not functional. But largest does not always mean better. However, if the result computed by All-Source-Or-Key-Vars is too large (e.g., in Example B.5), it is possible to obtain a smaller result by simply removing arguments from Skolem functors (and it is clear that removing is an easier activity than adding). It turns out also that Source-Here-and-Ref-Vars also provides the smaller result set. We believe that the various Skolem functors should contain at least the arguments identified by Source-Here-and-Ref-Vars. This poses a bound to what it is possible to remove from the arguments of Skolem functors.

C Other Examples

In this section we provide a number of more comprehensive examples, to understand the various steps defined in our schema mapping and query generation algorithms.
C.1 An Example About Query Generation

Example C.1 Consider the schema mapping problem depicted in Figure 10.

We first compute the following schema mapping, resorting to the basic or our improved novel schema mapping algorithm (subscripts added to implications will be useful next):

\[
P_3(p, n, e) \rightarrow_{pp} P_{2a}(p, n, e)
C_3(c, m) \rightarrow_{cc} C_{2a}(c, m, p')
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} C_{2a}(c, m, p), P_{2a}(p, n, e)
\]

We skolemize logical mappings, and then rewrite them as unitary logical mappings:

\[
P_3(p, n, e) \rightarrow_{pp} P_{2a}(p, n, e)
C_3(c, m) \rightarrow_{cc} P_{2a}(f_p(c), f_n(f_p(c)), f_e(f_p(c)))
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} P_{2a}(p, n, e)
C_3(c, m) \rightarrow_{cc} C_{2a}(c, m, f_p(c))
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} C_{2a}(c, m, p)
\]

We now check each individual unitary logical mapping for functionality. Let us first consider the fifth mapping:

\[
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} C_{2a}(c, m, p)
\]

To check its functionality over \(m\), we need to test emptiness of the following query:

\[
O_3(c, p), C_3(c, m), P_3(p, n, e), C_3(c', m'), P_3(p', n', e'),
c = c', m \neq m'
\]

This is guaranteed by the fact that \(c\) is a key also for the source relation \(C_3\).

To check its functionality over \(p\), we need to test emptiness of the following query:

\[
O_3(c, p), C_3(c, m), P_3(p, n, e), C_3(c', m'), P_3(p', n', e'),
c = c', p \neq p'
\]

This is guaranteed by the fact that \(c\) is a key for both source relations \(C_3\) and \(O_3\).
Let us now consider the second mapping:

$$C_3(c, m) \rightarrow_{ce} P_{2a}(f_p(c), f_n(f_p(c)), f_e(f_p(c)))$$

Note, in this case, that Skolem functors occur in the positions in the relational atom for $P_{2a}$.

To check functionality of this mapping over over $n$, we need to test emptiness of the following query:

$$C_3(c, m), C_3(c', m'), f_p(c) = f_p(c'), f_n(f_p(c)) \neq f_n(f_p(c'))$$

This query turns out to be unsatisfiable, since $f_p(c) = f_p(c')$ implies $f_n(f_p(c)) = f_n(f_p(c'))$.

Let us now consider the fourth mapping:

$$C_3(c, m) \rightarrow_{ce} C_{2a}(c, m, f_p(c))$$

To check functionality of this mapping over over $m$ and $p$, respectively, we need to test emptiness of the following queries:

$$C_3(c, m), C_3(c', m'), c = c', m \neq m'$$

$$C_3(c, m), C_3(c', m'), c = c', f_p(c) \neq f_p(c')$$

It turns out that both the above queries are always empty, also because $c$ is a key for relation $C_3$.

Functionality of the first and third mappings can be verified easily.

We now identify key conflicts.

We first consider the conflicting set for $P_{2a}$, that is, the logical mappings having $P_{2a}$ as target relation:

$$P_3(p, n, e) \rightarrow_{pp} P_{2a}(p, n, e)$$
$$C_3(c, m) \rightarrow_{ce} P_{2a}(f_p(c), f_n(f_p(c)), f_e(f_p(c)))$$
$$O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} P_{2a}(p, n, e)$$

The second mapping can not conflict with the other two mappings, since it is the only mapping generating invented values for the key of $P_{2a}$.

Moreover, the third mapping always generates a subset of the tuples generated by the first mapping. For example, the fact that these two mappings do not key conflict on $n$ can be checked by testing emptiness of the following query (over instances satisfying the source integrity constraints):

$$P_3(p, n, e), O_3(c', p'), C_3(c', m'), P_3(p', n', c'), p = p', n \neq n'$$

The case for $e$ is similar.

We then consider the conflicting set for relation $C_{2a}$:

$$C_3(c, m) \rightarrow_{cc} C_{2a}(c, m, f_p(c))$$
$$O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow_{oc} C_{2a}(c, m, p)$$

We first check for a possible key conflict on attribute $m$: it is possible that the two mappings generate tuples with a same value for $c$ (the key) but a different value for $m$? This is not the case, since the following query is unsatisfiable:

$$C_3(c, m), O_3(c', p'), C_3(c', m'), P_3(p', n', e'), c = c', m \neq m'$$
We then check for a possible key conflict on attribute \( p \), by considering the following query:

\[
C_3(c, m), O_3(c', p'), C_3(c', m'), P_3(p', n', c'), c = c', f_p(c, m) \neq p'
\]

The above query is satisfiable. A possible source instance leading to violation of the target key constraint would have the following form:

\[
C_3(c, m), O_3(c, p'), C_3(c, m), P_3(p', n', c')
\]

That is: there is a car \( c \) having an owner \( p' \). This is a soft key conflict.

In summary, there is only a soft key conflict on \( C_3, p \), between the following two mappings:

\[
\begin{align*}
C_3(c, m) & \rightarrow_{cc} C_{2a}(c, m, f_p(c)) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow_{oc} C_{2a}(c, m, p)
\end{align*}
\]

Since the second mapping is preferred to the first mapping, we can rewrite the first mapping as follows:

\[
C_3(c, m), \neg \phi_{oc}^c(c) \rightarrow_{cc} C_{2a}(c, m, f_p(c))
\]

where \( \phi_{oc}^c(c) \) is defined as

\[
\phi_{oc}^c(c) = \{ c \mid O_3(c, p'), C_3(c, m'), P_3(p', n', c') \}
\]

Now, the above rewriting should be performed as well on other unitary logical mappings derived from the same original logical mapping whose implication has been labeled \( cc \).

Overall, we obtain the following modified schema mapping:

\[
\begin{align*}
P_3(p, n, e) & \rightarrow_{pp} P_{2a}(p, n, e) \\
C_3(c, m), \neg \phi_{oc}^c(c) & \rightarrow_{cc} P_{2a}(f_p(c), f_n(f_p(c)), f_c(f_p(c))) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow_{oc} P_{2a}(p, n, e) \\
C_3(c, m), \neg \phi_{oc}^c(c) & \rightarrow_{cc} C_{2a}(c, m, f_p(c)) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) & \rightarrow_{oc} C_{2a}(c, m, p)
\end{align*}
\]

Finally, query generation is performed by reversing the implications, and reading the result as a nonrecursive Datalog program with skolem functors and stratified negation:

\[
\begin{align*}
P_{2a}(p, n, e) & \leftarrow P_3(p, n, e) \\
P_{2a}(f_p(c), f_n(f_p(c)), f_c(f_p(c))) & \leftarrow C_3(c, m), \neg OC_{tmp}(c) \\
P_{2a}(p, n, e) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \\
C_{2a}(c, m, f_p(c)) & \leftarrow C_3(c, m), \neg OC_{tmp}(c) \\
C_{2a}(c, m, p) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e) \\
OC_{tmp}(c) & \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)
\end{align*}
\]

Figure 11 shows a data transformation computed by the queries above. The invented person is somehow necessary, since attribute \( C_{2a} \_person \) (the car owner) is mandatory in the target schema. Note that the data transformation computed by basic schema mapping algorithms would be essentially the same shown in Figure 2 (in which \( P_2 \) and \( C_2 \) have to be replaced by \( P_{2a} \) and \( C_{2a} \), respectively); there, the target instance violates the key constraint on \( C_{2a} \) and contains an additional useless tuple in \( P_{2a} \).
C.2 An Example About Referenced-Attribute Correspondences

**Example C.2** Consider the mapping problem shown in Figure 12. In the source schema we have persons and cars; each car can have an owner and a driver. In the target schema we have cars, each car can be associated with the name of its owner (o-name) and driver (d-name).

We have the following source logical relations:

- $P_4(p, n, e)$
- $C_4(c, m)$
- $O_4(c, p), C_4(c, m), P_4(p, n, e)$
- $D_4(c, p), C_4(c, m), P_4(p, n, e)$

and the following target logical relations:

- $C_{od}(c, m, on, dn), on = null, dn = null$
- $C_{od}(c, m, on, dn), on = null, dn ≠ null$
- $C_{od}(c, m, on, dn), on ≠ null, dn = null$
After the initial pruning, there are neither subsumptions nor implications. Pruning based on
them. The first mapping is the least preferred (\(S\))

Skeletons/candidate logical mappings:

- **S1**: \(P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn = null\) / no correspondence covered
- **S2**: \(C_4(c,m) / C_{od}(c,m,od,dn)\), \(on = null, dn = null / c_c, c_m\)
- **S3**: \(O_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn = null / c_c, c_m\) — to be pruned, because the coverage degree for \(c_m\) is mand-null
- **S4**: \(D_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn = null / c_c, c_m\) — to be pruned, because the coverage degree for \(c_{dn}\) is mand-null
- **S5**: \(P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn \neq null / no correspondence covered\)
- **S6**: \(C_4(c,m) / C_{od}(c,m,od,dn)\), \(on = null, dn \neq null / c_c, c_m\) — to be pruned, because the coverage degree for attribute \(dn\) is none-null, and \(dn\) is not a foreign key
- **S7**: \(O_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn \neq null / c_c, c_m\) — to be pruned, because the coverage degree for \(c_m\) is mand-null
- **S8**: \(D_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on = null, dn \neq null / c_c, c_m, c_{dn}\)
- **S9**: \(P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn = null / no correspondence covered\)
- **S10**: \(C_4(c,m) / C_{od}(c,m,od,dn)\), \(on \neq null, dn = null / c_c, c_m\) — to be pruned, because the coverage degree for \(c_m\) is none-null, and \(on\) is not a foreign key
- **S11**: \(O_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn = null / c_c, c_m, c_{on}\)
- **S12**: \(D_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn = null / c_c, c_m\) — to be pruned, because the coverage degree for \(c_{dn}\) is mand-null
- **S13**: \(P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn \neq null / no correspondence covered\)
- **S14**: \(C_4(c,m) / C_{od}(c,m,od,dn)\), \(on \neq null, dn \neq null / c_c, c_m\) — to be pruned, because the coverage degree for attributes \(on, dn\) is none-null, and \(on, dn\) are not foreign keys
- **S15**: \(O_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn \neq null / c_c, c_m, c_{on}\) — to be pruned, because the coverage degree for attribute \(dn\) is none-null, and \(dn\) is not a foreign key
- **S16**: \(D_4(c,p), C_4(c,m), P_4(p,n,e) / C_{od}(c,m,od,dn)\), \(on \neq null, dn \neq null / c_c, c_m, c_{dn}\) — to be pruned, because the coverage degree for attribute \(on\) is none-null, and \(on\) is not a foreign key

After the initial pruning, there are neither subsumptions nor implications. Pruning based on
non-null extension is also not possible.

The resulting schema mapping is:

\[
\begin{align*}
C_4(c,m) & \rightarrow C_{od}(c,m,od',dn') \\
O_4(c,p), C_4(c,m), P_4(p,n,e) & \rightarrow C_{od}(c,m,n,od') \\
D_4(c,p), C_4(c,m), P_4(p,n,e) & \rightarrow C_{od}(c,m,od',n)
\end{align*}
\]

Each individual logical mapping is functional. However, the mappings do key conflict among
them. The first mapping is the least preferred (\(od'\) and \(dn'\) would correspond to null values).
The second mapping is preferred to the third one over attribute $on$, but the third mapping is preferred to the second one over attribute $dn$. By considering the above preferences, our query generation algorithm is then able to compute the following desirable transformation:

\[
\begin{align*}
C_{od}(c, m, null, null) & \leftarrow C_4(c, m), \neg OC_{tmp}(c), \neg DC_{tmp}(c) \\
C_{od}(c, m, n, null) & \leftarrow O_4(c, p), C_4(c, m), P_4(p, n, e), \neg DC_{tmp}(c) \\
C_{od}(c, m, null, n) & \leftarrow D_4(c, p), C_4(c, m), P_4(p, n, e), \neg OC_{tmp}(c), \\
C_{od}(c, m, n, n') & \leftarrow O_4(c, p), C_4(c, m), P_4(p, n, e), D_4(c', p'), C_4(c', m'), P_4(p', n', e'), c = c' \\
OC_{tmp}(c) & \leftarrow O_4(c, p), C_4(c, m), P_4(p, n, e) \\
DC_{tmp}(c) & \leftarrow D_4(c, p), C_4(c, m), P_4(p, n, e)
\end{align*}
\]

The above transformation is able to associate, to each car, the names of their eventual owner and driver. See, for example, the data transformation shown in Figure 13. To the best of our understanding, a similar mapping can not be specified using attribute correspondences only.

**Figure 13:** A data transformation for the mapping problem of Example C.2

**C.3 An Example About Source Nullable Attributes**

**Example C.3** Consider the schema mapping problem depicted in Figure 14. Source logical relations:
• $P_2(p, n, e)$
• $C_2(c, m, p), p = null$
• $C_2(c, m, p), p \neq null, P_2(p, n, e)$

Target logical relations:
• $P_3(p, n, e)$
• $C_3(c, m)$
• $O_3(c, p), C_3(c, m), P_3(p, n, e)$

Candidate logical mappings:
• $S_1 : P_2(p, n, e) / P_3(p, n, e) / p_1, p_2, p_3$
• $S_2 : P_2(p, n, e) / O_3(c, p), C_3(c, m), P_3(p, n, e) / p_1, p_2, p_3$
• $S_3 : C_2(c, m, p), p = null / C_3(c, m) / c_1, c_2$
• $S_4 : C_2(c, m, p), p = null / O_3(c, p'), C_3(c, m), P_3(p', n, e) / c_1, c_2$ — note that this skeleton does not cover $o_2$
• $S_5 : C_2(c, m, p), p \neq null, P_2(p, n, e) / C_3(c, m) / c_1, c_2$
• $S_6 : C_2(c, m, p), p \neq null, P_2(p, n, e) / P_3(p, n, e) / p_1, p_2, p_3$
• $S_7 : C_2(c, m, p), p \neq null, P_2(p, n, e) / O_3(c, p), C_3(c, m), P_3(p, n, e) / all six correspondences$

$S_1$ subsumes both $S_2$ and $S_6$. $S_3$ subsumes $4$. $S_7$ implies $S_5$.

Schema mapping:

\[
\begin{align*}
P_2(p, n, e) & \rightarrow P_3(p, n, e) \\
C_2(c, m, p), p = null & \rightarrow C_3(c, m) \\
C_2(c, m, p), p \neq null, P_2(p, n, e) & \rightarrow O_3(c, p), C_3(c, m), P_3(p, n, e)
\end{align*}
\]

Transformation:

\[
\begin{align*}
P_3(p, n, e) & \leftarrow P_2(p, n, e) \\
P_3(p, n, e) & \leftarrow C_2(c, m, p), p \neq null, P_2(p, n, e) \\
C_3(c, m) & \leftarrow C_2(c, m, p), p = null \\
C_3(c, m) & \leftarrow C_2(c, m, p), p \neq null, P_2(p, n, e) \\
O_3(c, p) & \leftarrow C_2(c, m, p), p \neq null, P_2(p, n, e)
\end{align*}
\]

Figure 15 shows a sample data transformation computed by the queries above. ■
C.4 An Example About Soft Key Conflict Resolution

**Example C.4** Source/target schemas:

\[
egin{align*}
S_1(k, a, b, c) \\
S_2(k, a, b, c) \\
S_3(k, a, b, c)
\end{align*}
\]

Schema mapping:

\[
egin{align*}
S_1(k_1, a_1, b_1, c_1) &\rightarrow_1 T(k_1, a_1, b_1', c_1') \\
S_2(k_2, a_2, b_2, c_2) &\rightarrow_2 T(k_2, a_2', b_2, c_2') \\
S_3(k_3, a_3, b_3, c_3) &\rightarrow_3 T(k_3, a_3', b_3', c_3)
\end{align*}
\]

Skolemized schema mapping (a different but equivalent skolemization is possible):

\[
egin{align*}
S_1(k_1, a_1, b_1, c_1) &\rightarrow_1 T(k_1, a_1, f_b^1(k_1), \text{null}) \\
S_2(k_2, a_2, b_2, c_2) &\rightarrow_2 T(k_2, f_a^2(k_2), b_2, \text{null}) \\
S_3(k_3, a_3, b_3, c_3) &\rightarrow_3 T(k_3, f_a^3(k_3), f_b^3(k_3), c_3)
\end{align*}
\]

Mapping rewrites:

\[
egin{align*}
k_1 = k_2, \neg S_2(k_2, a_2, b_2, c_2), \\
k_1 = k_3, \neg S_3(k_3, a_3, b_3, c_3) &\rightarrow_1 T(k_1, a_1, f_b^1(k_1), \text{null}) \\
S_2(k_2, a_2, b_2, c_2), \\
k_2 = k_1, \neg S_1(k_1, a_1, b_1, c_1), \\
k_2 = k_3, \neg S_3(k_3, a_3, b_3, c_3) &\rightarrow_2 T(k_2, f_a^2(k_2), b_2, \text{null}) \\
S_3(k_3, a_3, b_3, c_3), \\
k_3 = k_1, \neg S_1(k_1, a_1, b_1, c_1), \\
k_3 = k_2, S_2(k_2, a_2, b_2, c_2) &\rightarrow_3 T(k_3, f_a^3(k_3), f_b^3(k_3), c_3)
\end{align*}
\]

New mappings added:

\[
egin{align*}
S_1(k_1, a_1, b_1, c_1), \\
S_2(k_2, a_2, b_2, c_2), \\
k = k_1, k = k_2, \\
k = k_3, \neg S_3(k_3, a_3, b_3, c_3) &\rightarrow_{1,2} T(k, a_1, b_2, \text{null})
\end{align*}
\]

Figure 15: A data transformation for the mapping problem of Example C.3
\[ S_1(k_1, a_1, b_1, c_1), \]
\[ S_3(k_3, a_3, b_3, c_3), \]
\[ k = k_1, k = k_3, \]
\[ k = k_2, \neg S_2(k_2, a_2, b_2, c_2) \quad \neg_{1,3} T(k, a_1, f^1_3(k), c_3) \]
\[ S_2(k_2, a_2, b_2, c_2), \]
\[ S_3(k_3, a_3, b_3, c_3), \]
\[ k = k_2, k = k_3, \]
\[ k = k_1, \neg S_1(k_1, a_1, b_1, c_1) \quad \neg_{2,3} T(k, f^2_3(k), b_2, c_3) \]
\[ S_1(k_1, a_1, b_1, c_1), \]
\[ S_2(k_2, a_2, b_2, c_2), \]
\[ S_3(k_3, a_3, b_3, c_3), \]
\[ k = k_1, k = k_2, k = k_3 \quad \neg_{1,2,3} T(k, a_1, b_2, c_3) \]