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

Luca Cabibbo
Università Roma Tre
cabibbo@dia.uniroma3.it

ABSTRACT

We consider the following scenario for a mapping system: given a source schema, a target schema, and a set of value correspondences between these two schemas, 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. In this paper, 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 [2, 6]. 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 [2, 17], including both research prototypes, such as Clio [13, 15], and commercial industry tools, such as Altova MapForce, IBM Rational Data Architect, Microsoft BizTalk Mapper, and Stylus Studio Mapper.

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 [15]: 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 [4]. 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 to cope with relational schemas [13], have then been extended to deal also with XML nested data [15]. Further research has then mainly focused on improving the translation between XML data (e.g., [5, 17]). 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., [18]), foreign keys (e.g., as inclusion dependencies [13, 15]), and nullable attributes (e.g., [15, 18]) 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 [18, 4]) 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 tra-
ditional attribute correspondences [13, 15]. They permit to
express more precise mappings, while retaining a simple and
intuitive semantics.

Our algorithms can be applied to mapping problems in-
volving relational schemas with primary keys, foreign keys
(used to reference simple keys and forming a weakly acyclic
set [4]), and nullable attributes. As output, our algorithms
generate transformations expressed as non-recursive Data-
log queries, with Skolem functors and safe (stratified) nega-
tion [1, 7].

**Motivating Examples.** Our first example shows ben-
efits that can be gained by managing keys, foreign keys and
nullable attributes in relational mappings.

Example 1. Consider the mapping problem graphically
depicted in Figure 1. The schemas comprise keys (shown under-
lined), 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 correspond-
dences, visually depicted as solid arrows. It involves two car
registration databases: CARS3, the source schema, shown
on the left, and CARS2, the target schema, shown on the
right. Attributes person and car represent person and car
identifiers, respectively.

![Figure 1: A sample mapping problem](image)

For this mapping problem, basic schema mapping gener-
ation algorithms [13, 15] (which do not fully consider null-
able attributes) compute the following schema mapping (for
space reasons, we denote relation and attribute names only
by their initial letters):

\[
P_3(p, n, e) \rightarrow P_2(p, n, e)
\]

\[
O_4(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
\]

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 [13, 15]
(which don’t consider key constraints) compute the following
transformation from CARS3 to CARS2 (expressed as a non-
recursive Datalog program with Skolem functors):

\[
P_3(p, n, e) \leftarrow P_3(p, n, e)
\]

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

\[
C_2(c, m, f_P(c, m)) \leftarrow C_3(c, m)
\]

\[
P_2(f_P(c, m), f_X(c, m), f_E(c, m)) \leftarrow C_3(c, m)
\]

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 1. (cont.) A more natural and desirable data
transformation for the mapping problem of Example 1 is
shown in Figure 3. For the same mapping problem, our novel
schema generation algorithm (which also considers nullable
attributes) computes the following schema mapping:

\[
P_3(p, n, e) \rightarrow P_2(p, n, e)
\]

\[
O_4(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
\]

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

Here, 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

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.

Figure 2 shows a data transformation computed by the
queries above. (The source and target instances are shown,
respectively, at top and bottom of the figure. Symbols π, η, and ϵ denote invented values.) Overall, the above trans-
formation has the following drawbacks: First, for each car
having an owner, the program generates in relation C_2 a t-
uple for the car with its correct owner, plus a further tuple
for the same car with an invented owner. This additional
tuple is not required; it also leads to a violation of the key
constraint on C_2. The invented person is represented by an
additional tuple in P_2; this tuple is also undesirable. Second,
for each car not having an owner, the program generates in
C_2 a tuple for the car with an invented owner. However,
for a car without an owner, a better (and admissible) solu-
tion would consist in generating just a tuple in C_2 in which
attribute person is set to null.

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 1. (cont.) A more natural and desirable data
transformation for the mapping problem of Example 1 is
shown in Figure 3. For the same mapping problem, our novel
schema generation algorithm (which also considers nullable
attributes) computes the following schema mapping:

\[
P_3(p, n, e) \rightarrow P_2(p, n, e)
\]

\[
O_4(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
\]

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

Here, 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

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.

Figure 2 shows a data transformation computed by the
queries above. (The source and target instances are shown,
respectively, at top and bottom of the figure. Symbols π, η, and ϵ denote invented values.) Overall, the above trans-
formation has the following drawbacks: First, for each car
having an owner, the program generates in relation C_2 a t-
uple for the car with its correct owner, plus a further tuple
for the same car with an invented owner. This additional
tuple is not required; it also leads to a violation of the key
constraint on C_2. The invented person is represented by an
additional tuple in P_2; this tuple is also undesirable. Second,
for each car not having an owner, the program generates in
C_2 a tuple for the car with an invented owner. However,
for a car without an owner, a better (and admissible) solu-
tion would consist in generating just a tuple in C_2 in which
attribute person is set to null.

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 1. (cont.) A more natural and desirable data
transformation for the mapping problem of Example 1 is
shown in Figure 3. For the same mapping problem, our novel
schema generation algorithm (which also considers nullable
attributes) computes the following schema mapping:

\[
P_3(p, n, e) \rightarrow P_2(p, n, e)
\]

\[
O_4(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
\]

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

Here, 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

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.

Figure 2 shows a data transformation computed by the
queries above. (The source and target instances are shown,
respectively, at top and bottom of the figure. Symbols π, η, and ϵ denote invented values.) Overall, the above trans-
formation has the following drawbacks: First, for each car
having an owner, the program generates in relation C_2 a t-
uple for the car with its correct owner, plus a further tuple
for the same car with an invented owner. This additional
tuple is not required; it also leads to a violation of the key
constraint on C_2. The invented person is represented by an
additional tuple in P_2; this tuple is also undesirable. Second,
for each car not having an owner, the program generates in
C_2 a tuple for the car with an invented owner. However,
for a car without an owner, a better (and admissible) solu-
tion would consist in generating just a tuple in C_2 in which
attribute person is set to null.

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 1. (cont.) A more natural and desirable data
transformation for the mapping problem of Example 1 is
shown in Figure 3. For the same mapping problem, our novel
schema generation algorithm (which also considers nullable
attributes) computes the following schema mapping:

\[
P_3(p, n, e) \rightarrow P_2(p, n, e)
\]

\[
O_4(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p), P_2(p, n, e)
\]

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

Here, 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

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.

Figure 2 shows a data transformation computed by the
queries above. (The source and target instances are shown,
respectively, at top and bottom of the figure. Symbols π, η, and ϵ denote invented values.) Overall, the above trans-
formation has the following drawbacks: First, for each car
having an owner, the program generates in relation C_2 a t-
uple for the car with its correct owner, plus a further tuple
for the same car with an invented owner. This additional
tuple is not required; it also leads to a violation of the key
constraint on C_2. The invented person is represented by an
additional tuple in P_2; this tuple is also undesirable. Second,
for each car not having an owner, the program generates in
C_2 a tuple for the car with an invented owner. However,
for a car without an owner, a better (and admissible) solu-
tion would consist in generating just a tuple in C_2 in which
attribute person is set to null.

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.
from \textit{CARS}_1 to \textit{CARS}_2 (expressed as a non-recursive Database program with stratified negation):

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

Here, the second rule deals only with cars having an owner; the third rule defines an intermediate relation \(OC_{tmp}\) (for \textit{Owned Cars}) and the fourth rule deals only with cars without an owner (it assigns to them a \textit{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 algorithm for schema mapping generation takes care of nullable attributes, to avoid the generation of useless tuples made of invented and null values only. Furthermore, our query generation algorithm takes into account key constraints, to 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.

\textbf{Example 2.} Consider the mapping problem shown in Figure 4. The source schema \textit{CARS}_3 is as in Example 1. The target schema \textit{CARS}_1 consists of just a single relation, intended to store a tuple for each car, possibly with the name of its owner (or \textit{null}, otherwise).

\textbf{Figure 4: Another sample mapping problem}

For this mapping problem, even our mapping algorithms generate the following (incorrect) transformation:

\[
\begin{align*}
C_1(c,m,\text{n}) & \leftarrow O_3(c,p), C_3(c,m), P_3(p,n,e) \\
OC_{tmp} & \leftarrow O_3(c,p), C_3(c,m), P_3(p,n,e) \\
C_1(c,m,\text{null}) & \leftarrow C_3(c,m), \neg OC_{tmp}(c) \\
C_1(f_3(p),f_M(p),\text{n}) & \leftarrow P_3(p,n,e)
\end{align*}
\]

Figure 5 shows a data transformation computed by the queries above. This is indeed the natural and desired transformation.

\textbf{Figure 5: A data transformation for Example 2}

the intended mapping. Indeed, attribute correspondence labeled \textit{cn} in Figure 4 specifies that each value occurring in attribute \textit{name} in the source should occur as well in attribute \textit{name} in the target database. However, in the desired mapping, we would like that only names of car owners occur in attribute \textit{name} in the target database.

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

\textbf{Example 2. (cont.)} For the same mapping problem of Example 2, we would like to let attribute \textit{name} in the target database correspond to names of car owners, that is, to those values occurring in the source database as \textit{name} in tuples of relation \(P_3\) that can be reached by navigating the foreign key from \(O_3\text{.person} \rightarrow P_3\). This can be specified by using, instead of the traditional attribute correspondence \textit{cn}, the following referenced-attribute correspondence (symbol \textit{←} denotes, intuitively, the “traversal” of a foreign key):

\[
cn' : (O_3\text{.person} \leftarrow P_3\text{.name}, C_1\text{.name})
\]

Using these correspondences, our algorithms are indeed able to generate the following correct transformation:

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

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

\textbf{Figure 6: A better data transformation for the mapping problem of Example 2}

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

For space reasons, a number of detailed definitions, procedures, and examples have been omitted; they are available in the full version of this paper [3].

2. PRELIMINARIES

2.1 Relational Model

A relation schema \( R(A_1, \ldots, A_n) \) is a named set of attributes. A relational schema \( R = \{ R_1, \ldots, R_n \} \) is a set of relation schemas, together with a set \( \Gamma_R \) of integrity constraints (described next). 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 [4], something that we will assume in the remainder of the paper.

2.2 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 [13] and then refined in [15, 5], 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 \((R_1, A_1, A_2)\), 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 correspondence 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 represent 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 \((s \rightarrow t \mathrm{tg})\) [4], 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 [13, 15] 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 \( s \rightarrow t \mathrm{tg} \) of the form \( \forall T_1 
\]
tions 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:

$$P_{2a}(p, n, e) \Rightarrow P_3(p, n, e)$$
$$C_{2a}(c, m, p), P_{2a}(p, n, e) \Rightarrow O_3(c, p), C_3(c, m), P_3(p, n, e)$$

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:

$$P_3(p, n, e) \leftarrow P_{2a}(p, n, e)$$
$$P_3(p, n, e) \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e)$$
$$C_3(c, m) \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e)$$
$$O_3(c, p) \leftarrow C_{2a}(c, m, p), P_{2a}(p, n, e)$$

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 [15]. Because of this, the resulting transformations are, in general, non-recursive Datalog programs with Skolem functions, where Skolem functions are the mechanism used to specify “invented values” that should occur in the target instance [7].

3. REFERENCED-ATTRIBUTE CORRESPONDENCES

Recall [13, 15] that an attribute correspondence $(A, B)$ 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, $A$ are names of persons, while $B$ are names of car owners, where the latter are 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 [13]. 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 correspondences, called referenced-attribute correspondences, which generalize attribute correspondences.

A referenced attribute is an expression $R_1.A_1 \cdots \cdots R_n.A_n$, 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 example, in schema $CARS_3$ of Example 2, referenced attribute $O_3.person \rightsquigarrow P_3.name$ 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, $(O_3.person \rightsquigarrow P_3.name, C_1.name)$. 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 $R_1.A_1 \cdots \cdots R_n.A_n$ 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, for $1 \leq i < n$, equality conditions $t(R_i.A_i) = t(R_{i+1}.K_{i+1})$, 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 $(T_1, T_2)$. 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 s-t tgd of the form $\forall T_1 \exists T_2.V$, where $V$ denotes the conjunction of a set of conditions, comprising, for each covered referenced-attribute correspondence $(R_1.A_1 \cdots \cdots R_n.A_n, R_1'.A_1' \cdots \cdots R_m'.A_m')$, a condition $t_n = t_{n'}$, where $t_n, t_{n'}$ are the terms occurring in the positions respectively corresponding to referenced attributes $A_n, A_{n'}$.

Example 3. Figure 8 shows a mapping problem involving a referenced-attribute correspondence. This is a variant of the problem considered in Example 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.

For this mapping problem, current mapping algorithms, modified with the above definition of r-a correspondence and the related notion of coverage, are able to compute the fol-

![Figure 8: A referenced-attribute correspondence](image)
following schema mapping:

\[ O_2(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_{1a}(c, m, n) \]

\[ C_3(c, m) \rightarrow C_{1a}(c, m, n') \]

This is the best possible schema mapping for this problem. 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 5) 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):

\[ C_{1a}(c, m, n) \leftarrow O_2(c, p), C_3(c, m), P_3(p, n, e) \]

\[ OC_{tmp}(c) \leftarrow O_2(c, p), C_3(c, m), P_3(p, n, e) \]

\[ C_{1a}(c, m, f_N(c)) \leftarrow C_3(c, m), \neg OC_{tmp}(c) \]

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 exists 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 correspondence, and then selecting, in each schema, a path of foreign keys to the relation containing the referenced attribute.

4. 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 extend the basic relational schema mapping generation algorithm outlined in Section 2.2 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]).

4.1 Logical Relation Generation

Logical relations are computed separately in each individual schema. In basic logical relation generation, each logical relation (or tableau) is computed by applying a standard chase procedure to each individual (base) relation of a schema. 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 \( A = null \) nor \( A \neq null \) 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 \( x \neq y \) \( \notin T \), and \( \bot \) otherwise.

ind rule 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 \( u.x \neq null \). 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 relational mapping generation algorithm: (i) a nonnull 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 [4]), and nullable attribute constraints.

Example 4. Consider the target schema $\text{CARS}_2$ shown in Figure 1. Logical relations for schema $\text{CARS}_2$ are the following (partial tableaux can still be considered as “autonomous concepts” or “representative instances” of a schema): (i) $\text{P}_2(p, n, e)$: persons (unrelated to cars); (ii) $\text{C}_2(c, m, p), p = \text{null}$: cars without an owner; and (iii) $\text{C}_2(c, m, p), p \neq \text{null}, \text{P}_2(p, n, e)$: owned cars, with their owner.

If we apply our modified chase procedure to a base relation $R$, in general we obtain a set $\mathcal{T}_R = \{T_1, \ldots, T_n\}$ 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 $\mathcal{T}_R$ are partially disjoint, that is, $T_i \cap T_j = \emptyset$ if $i \neq j$ (over databases satisfying the schema constraints); and (ii) $R = T_1 \cup \ldots \cup T_n$.

### 4.2 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. (A number of conclusions in this section are motivated by a case-by-case reasoning, described in the full version of the paper [3].)

**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:

- **nonnull**, 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, the coverage level of a referenced attribute $R_1.A_1 \leadsto \ldots \leadsto R_n.A_n$ in $T$ is defined as the coverage level of $R_n.A_n$, provided that all previous attributes $R_1.A_1, \ldots, R_{n-1}.A_{n-1}$ in the path are covered 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, this definition applies also to 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 (null, mand).

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.** 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; (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:

- there is a correspondence in $V$ having coverage degree (nonnull, null), (mand, null), or (null, nonnull) — this rule is motivated by the fact that, if there exists such a candidate logical mapping $m$, by the logical relation generation algorithm, there should also be a different but preferable candidate logical mapping $m'$;
- 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 $A_2$).

**Pruning Based on Subsumption and on Implication.** As in the basic logical mapping generation, subsumed and implied mappings should be pruned [13, 15].

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: (i) the relational atoms of $T'$ are a superset of the relational atoms in $T$; (ii) the constraint atoms (conditions) in $T'$ are also a superset of those in $T$ or they imply them; (iii) the null
conditions in $T'$ are also a superset of those in $T$; and (iv) the non-null conditions in $T'$ are also a superset of those in $T$ (in each point, possibly after some renaming of variables). Moreover, $T'$ is a strict sub-tableau of $T$ (written $T' \prec T$) 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 $V = V'$. We prune $m'$ since it covers the same set of correspondences that are covered by the “smaller” (and more general) 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 $T_2$ is a sub-tableau of $T_2$ (this implies $V \subseteq V'$). 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).

**Pruning Based on Non-Null Extension.** First, note that each partial tableaux can be depicted as a rooted graph (or a rooted tree, if foreign keys are acyclic), in which nodes represent relational atoms, arcs represent traversals of foreign keys, and the 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 4.1). 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, with respect to partial tableaux over schema $\text{CARS}_2$ of Example 4, 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.

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 ($V \subset V'$), candidate logical mapping $m$ should be pruned.

This pruning rule aims at avoiding the generation, in the target instance, of “useless tuples” made of null and invented values only (for case (a)) or containing tuples stating partially duplicate and less informative facts (for case (b)).

**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 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 2.2 and 3.

**Example 5.** Consider again the mapping problem of Example 1, depicted in Figure 1.

There are seven candidate logical mappings, as follows:

S1: $P_3(p, n, e) / P_2(p, n, e) / p_1, p_2, p_3$

S2: $O_3(c, p), C_3(c, m), P_3(p, n, e) / P_2(p, n, e) / p_1, p_2, p_3$

S3: $C_3(c, m) / C_2(c, m, p), p = \text{null} / c_1, c_2$

S4: $O_3(c, p), C_3(c, m), P_3(p, n, e) / C_2(c, m, p), p = \text{null} / c_1, c_2, o_1$

S5: $C_3(c, m) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / c_1, c_2$

S6: $P_3(p, n, e) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / p_1, p_2, p_3$

S7: $O_3(c, p), C_3(c, m), P_3(p, n, e) / C_2(c, m, p), p \neq \text{null}, P_2(p, n, e) / p_1, p_2, p_3, c_1, c_2, o_1, o_2$

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 $S_5$). $S_2$ 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 $S_4$).

Therefore, the computed schema mapping is the following:

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

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

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

**4.3 Discussion**

Algorithm 1 summarizes our schema mapping generation procedure. We have underlined differences with respect to the baseline algorithm described in Section 2.2.

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

**Input:** Source schema $\mathbf{S}$, with constraints $Γ_S$; target schema $\mathbf{T}$, with constraints $Γ_T$; set $\mathbf{C}$ 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 (use modified chase procedure) individual relations in $\mathbf{S}$ and $\mathbf{T}$ with $Γ_S$ and $Γ_T$, respectively.

2. **Identify Candidate Logical Mappings.** For each skeleton $(T_1, T_2)$, compute 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 imply a candidate logical mapping. 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 $Σ$ of logical mappings, each a source-to-target tuple-generating dependency [4] of the form:

$$∀x (φ_S(x) \rightarrow (∃y ψ_T(x, y)),$$

where $φ_S$ is a conjunctive query over the source schema, possibly with null and non-null conditions, and $ψ_T$ is a conjunctive query over the target schema. We call $φ_S$ and $ψ_T$, 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.
5. QUERY GENERATION — WITH KEYS

We now extend the basic relational query generation algorithm outlined in Section 2.2. The main extension consists in an additional intermediate processing step, concerning the management of key constraints. This step is positioned after mapping skolemization and before actual query generation.

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 consequence 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 1 and 5, depicted in Figure 1.

Logical Mapping Skolemization and Rewriting. We first skolemize logical mappings, as follows. Let \( m \) be a logical mapping, and \( y \) be an existentially quantified variable occurring in the consequent of \( m \). If \( y \) occurs only in a position for a nullable attribute, we replace \( y \) with \( \text{null} \). Otherwise, \( y \) occurs at least once in a position for a mandatory attribute. In this case, we skolemize \( y \) by replacing all occurrences of \( y \) with a new Skolem functor \( f_{m,y}(w) \), as follows. A different Skolem function \( f_{m,y} \) is used for each different logical mapping \( m \) and existentially quantified variable \( y \). Moreover, \( w \) is a set of terms chosen as follows. With logical mappings computed by our schema mapping generator, there are only two cases: (i) \( y \) is bound only to a key attribute (in the relational atom for the “root” relation of the target tableau); or (ii) \( y \) is bound to a non-key attribute (note that, in this case, \( y \) can also be bound to a key attribute). In case (i), \( w \) consists of all universally quantified variables occurring in logical mapping \( m \) (or, equivalently, of the variable(s) bound to the key attribute(s) of the “root” relation of the source tableau). In case (ii), \( w \) consists of the term(s) bound to the key attribute(s) of the single relational atom in which \( y \) occurs in a non-key attribute position. (Note that this can lead to nested Skolem terms.)

We use a skolemization procedure different from the one adopted by [13, 15] and described in Section 2.2, motivated by the goal of managing target key constraints and the consequent need to possibly obtain functional mappings.

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. Consider the logical mappings of Example 5. By rewriting them, we obtain the following "unitary" logical mappings:

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

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

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 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 \( m = \overline{\psi}_{i,j}(x) \rightarrow \overline{\psi}_i(x) \) 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 \( \phi^{k,v}_i(k,v) \) be the projection of \( \phi_i(x) \) over \( k, v \). Then, \( m \) is functional if, for each such variable \( v \), \( \phi^{k,v}_i(k,v) \wedge \phi^{k',v'}_i(k',v') \wedge k = k' \wedge v \neq v' \) is unsatisfiable over instances satisfying the source schema constraints. (Otherwise, we signal an error and stop.)

A minor modification in the procedure is needed to consider composite keys and Skolem functors. (See the full paper for more details [9].)

It turns out that the functionality check can be reduced to an 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). Decidability of such a check is a direct consequence of classical results [9, 10].

Example 7. Every unitary logical mapping of Example 6 is functional. In particular, functionality of the third mapping is guaranteed by the fact that \( \text{car} \) 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 \( m = \overline{\psi}_{i,j}(x) \rightarrow \overline{\psi}_i(x) \) and \( m' = \overline{\psi}_{i,j}(x) \rightarrow \overline{\psi}_j(x) \) 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 \( \phi^{k,v}_i(k,v) \wedge \phi^{k,v'}_j(k',v') \wedge k = k' \wedge v \neq v' \) is unsatisfiable. If the above expression is satisfiable for some \( v \), then we say that \( m \) and \( m' \) are key conflicting over \( v \). This check is also decidable.

Not every key conflict causes a real problem. A key conflict between a group of logical mappings is 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 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 8. Consider again the unitary logical mappings of Example 6. The first and fourth mappings generate tuples over the 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 \( \text{person} \). Indeed, the third logical mapping generates tuples for cars having an owner, with the actual owner in position \( \text{person} \), while the second logical mapping generates tuples for all cars, always with a null value in position \( \text{person} \). This key conflict is soft. Note also that these two logical mappings are not key conflicting over attribute \( \text{model} \).

**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; and (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. This is consistent with our position stated at the beginning of Section 4.

Let \( \Sigma \) be the set of unitary skolemized logical mappings obtained from logical mapping skolemization and rewriting. 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 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 two different unitary logical mappings \( m, m' \) in \( CS_R \). Moreover, we write \( \text{key}(R) \) to denote the key attribute of relation \( R \), and \( v \) to denote a non-key attribute of \( R \). We also write \( m \triangleright_v m' \) (or \( m' \triangleleft_v m \)), if there is a soft key conflict on \( v \) between \( m \) and \( m' \), and \( m \) is preferable to \( m' \) over \( v \). (E.g., \( m \) copies source values and \( m' \) invents new values for \( v \).) Relationship \( \triangleright_v \) is computed during key conflict identification.

The basic key conflict resolution is as follows. 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:

\[
\text{preferableTo}(m) = \{m' \in CS_R | \exists v : m' \triangleright_v m\}.
\]

If \( \text{preferableTo}(m) \) is not empty, we rewrite \( m \) by adding to its premise, with a conjunction, \( k = k' \land \neg \phi_{\text{key}(R)}^k(k') \) for each unitary logical mapping \( m' \in \text{preferableTo}(m) \), where \( \phi_{\text{key}(R)}^k(k') \) 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 \( \tilde{k} = k' \land \neg \phi_{\text{key}(R)}^{\tilde{k}}(k') \) to the premise of each other unitary logical mapping \( \hat{m} \) that has been obtained from the same “original” logical mapping in \( \Sigma \) as \( m \). (\( \tilde{k} \) 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 9. Consider again the soft key conflict on \( \text{person} \) identified in Example 8. 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. 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:

\[
C_3(c, m), \neg \phi_3^e(c) \rightarrow C_2(c, m, \text{null})
\]

\[
O_3(c, p), C_3(c, m), P_3(p, n, e) \rightarrow C_2(c, m, p)
\]

where \( \phi_3^e(c) = \{ c | O_3(c, p'), C_3(c, m'), P_3(p', n', e') \} \).

The above basic resolution procedure is able to deal with mappings that are key conflicting among them in a simple way. 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. To deal with this case, we need an additional step, as follows.

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 \( \text{key}(R) \) in mapping \( m_i \).

- The consequent of \( m_M \) is an atom \( R(k, t_1, \ldots, t_k) \) 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 in the full version [3].)

Moreover, let \( \text{preferableTo}(M) \) 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. If \( \text{preferableTo}(M) \) is not empty, we rewrite mapping \( m_M \) by...
adding \( k = k' \land \neg \psi_1^k \) to its premise, for each unitary logical mapping \( m' \in \text{preferableTo}(M) \), where \( \psi_1^k \) is the projection of the premise of \( m' \) on \( \text{key}(R) \), and \( k \) is the variable bound to \( \text{key}(R) \) in \( m_M \).

Example 10. Let \( m_1, m_2 \) be two soft key conflicting mappings (let \( k \) be the key attribute of \( R \)), as follows:

\[
m_1 : \psi_1^k(a, b, c) \rightarrow R(k, a, b, c)\\
m_2 : \psi_2^k(a, c) \rightarrow R(k, a, b, c)
\]

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 as follows:

\[
\phi_1^k(a, b, c) \land \neg \psi_2^k(k) \rightarrow R(k, a, b, c)\\
\phi_2^k(a, c) \land \neg \psi_1^k(k) \rightarrow R(k, a, b, c)
\]

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

\[
\phi_1^k(a, b, c) \land \phi_2^k(a', c') \land k = k' \rightarrow R(k, a, b', c')
\]

**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. This 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 stratified negation.

Example 11. Consider again the mapping problem of Examples 1, 5, 6, and 9. The current mapping consists of the following logical mappings:

\[
P_3(p, n, e) \quad O_3(c, p), C_3(c, m), P_3(p, n, e) \quad C_3(c, m), \\
\neg \{c \mid O_3(c, p'), C_3(c, m'), P_3(p', n', e') \} \quad C_2(c, m, null) \\
O_3(c, p), C_3(c, m), P_3(p, n, e) \quad C_2(c, m, p)
\]

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

\[
P_3(p, n, e) \leftarrow P_3(p, n, e)\\
P_3(p, n, e) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)\\
O_{C_3mp}(c) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)\\
C_2(c, m, null) \leftarrow C_3(c, m), \neg O_{C_3mp}(c)\\
C_2(c, m, p) \leftarrow O_3(c, p), C_3(c, m), P_3(p, n, e)
\]

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.

### 5.1 Discussion

Algorithm 2 summarizes our query generation generation procedure. We have underlined differences with respect to the baseline algorithm described in Section 2.2.

**Algorithm 2 (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, 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.

### 6. RELATED WORK

We use the Clio framework [13, 15] as our reference baseline, as many other mapping algorithms do (e.g., [5, 16]). 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 [13, 15], are able to deal with foreign keys, in the form of inclusion dependencies. The work in [15] and [18] consider nullable attributes, but not the significant case in which foreign keys can originate from nullable attributes. [18] 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. Indeed, some mapping systems deal with issues similar to ours, but at different stages of the mapping process. For example, [8] deals with duplicate elimination.
(somewhat similar to key conflict resolution) during query execution phase. Data exchange algorithms [4, 11] 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 [13]; 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 [16]. 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.

7. CONCLUSIONS

In this paper, we extended the original schema mapping and query generation algorithms proposed by Clio [13, 15] 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 [14]. 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 [12].

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

Several authors (see, e.g., [18, 4]) 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 mapping systems.

8. REFERENCES