Declarative Entity Resolution via Matching Dependencies and Answer Set Programs

Zeinab Bahmani and Leopoldo Bertossi
Carleton University, Ottawa, Canada.
zbahmani@connect.carleton.ca, bertossi@scs.carleton.ca

and

Solmaz Kolahi and Laks V. S. Lakshmanan
University of British Columbia, Vancouver, Canada.
{solmaz,laks}@cs.ubc.ca

Abstract
Entity resolution (ER) is an important and common problem in data cleaning. It is about identifying and merging records in a database that represent the same real-world entity. Recently, matching dependencies (MDs) have been introduced and investigated as declarative rules that specify ER. An ER process induced by MDs over a dirty instance leads to multiple clean instances, in general. In this work, we present disjunctive answer set programs (with stable model semantics) that capture through their models the class of alternative clean instances obtained after an ER process based on MDs. With these programs, we can obtain clean answers to queries, i.e. those that are invariant under the clean instances, by skeptically reasoning from the program. We investigate the ER programs in terms of expressive power for the ER task at hand. As an important special and practical case of ER, we provide a declarative reconstruction of the so-called union-case ER methodology, as presented through a generic approach to ER (the so-called Swoosh approach).

1 Introduction
Entity resolution (ER) is a classical, common and difficult problem in data cleaning. It deals with identifying and merging database records in a database that refer to the same real-world entity [Bleiholder and Naumann 2008; Elmagarmid, Ipeirotis and Verykios 2007]. In this way, duplicates are eliminated via a matching process. Matching dependencies (MDs) are declarative rules that generalize entity resolution tasks. They assert in declarative terms that certain attribute values in relational tuples have to be matched, i.e. made the same, when certain similarities hold between possibly other attribute values in those tuples. MDs were first introduced in [Fan 2008; Fan et al. 2009].

Example 1. Consider the relational schema \( R = \{ R(A, B) \} \), with a predicate \( R \) with attributes \( A \) and \( B \). The following symbolic expression
\[
R[A] \approx R[A] \rightarrow R[B] \doteqdot R[B],
\]
is an MD requiring that, if for any two database tuples \( R(a_1, b_1), R(a_2, b_2) \) in an instance \( D \) of the schema, when the values for attributes \( A \) are similar, i.e. \( a_1 \approx a_2 \), then their values for attribute \( B \) have to be made equal (matched), i.e. \( b_1 \) or \( b_2 \) (or both) have to be changed to a value in common.

Let us assume that \( \approx \) is reflexive and symmetric, and that \( a_2 \approx a_3 \), but \( a_2 \not\approx a_1 \not\approx a_3 \). The table on the left-hand side (LHS) below provides the extension for predicate \( R \) in \( D \). In it some duplicates are not resolved, since, e.g. the tuples (with tuple identifiers) \( t_4 \) and \( t_2 \) have similar – actually equal – values for attribute \( A \), but the values for \( B \) are different.

<table>
<thead>
<tr>
<th>( R(D) )</th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>( a_1 )</td>
<td>( b_1 )</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( a_1 )</td>
<td>( b_3 )</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>( a_2 )</td>
<td>( b_3 )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( a_2 )</td>
<td>( b_4 )</td>
</tr>
</tbody>
</table>

\( D \) does not satisfy the MD, and is a dirty instance. After applying the MD, we could get the instance \( D' \) on the right-hand side (RHS), where values for \( B \) have been identified. In principle, nothing prevents us from choosing a new value \( b_5 \) from the domain to do the matching. \( D' \) is stable in the sense that the MD holds in the traditional sense of an implication on \( D' \). We call \( D' \) a clean instance. In general, for a dirty instance and a set of MDs, multiple clean instances may exist. Notice that if we add the MD \( R[B] \approx R[B] \rightarrow R[A] \doteqdot R[A] \), creating a set of interacting MDs, a matching based on one MD may create new similarities that could enable a different MD in the set.

A dynamic semantics for MDs was introduced in [Fan et al. 2009], that requires a pair of instances: a first one where the similarities hold and second one where the matchings are enforced, like \( D \) and \( D' \) in Example 1. The MDs, as introduced in [Fan et al. 2009], do not specify how to match values. As we did in the example, we could even pick up a new value, e.g. \( b_5 \) above, for the value in common. This semantics was refined and extended in [Bertossi, Kolahi and Lakshmanan 2011] (cf. also [Bertossi, Kolahi and Lakshmanan 2012]), using a matching function (MF) to guide the matchings, for each of the participating attribute domains. The MFs induce a lattice-theoretic structure on the latter [Bertossi, Kolahi and Lakshmanan 2011]. An alternative dynamic semantics was introduced in [Gardezi, Bertossi and Kiriniga 2011]. It does not appeal to MFs, but matchings have to be mandatory (as also in [Bertossi, Kolahi and Lakshmanan 2011]) and minimal, i.e. a minimum number of
changes to attribute values is applied to satisfy the MDs.

In this work we revisit the approach to ER via MDs introduced in [Bertossi, Kolahi and Lakshmanan 2011], that uses MFs. In that scenario, a "dirty" instance \( D \) w.r.t. a set \( \Sigma \) of MDs may lead to several different clean and stable solutions \( D' \), each of which can be obtained by means of a provably terminating, but non-deterministic, chase-like procedure [Bertossi, Kolahi and Lakshmanan 2011]. The latter involves enforcing MDs iteratively by means of applying MFs. The set of all such clean instances is denoted by \( C(D, \Sigma) \). Our goal in this paper is to provide a declarative specification to this procedural data cleaning semantics.

In [Bertossi, Kolahi and Lakshmanan 2011], the clean answers to a query were introduced, as those that are (essentially) certain, i.e. true of all the clean instances (cf. Section 2 for details). They are invariant across the class \( C(D, \Sigma) \), and then are intrinsically "clean". The problem of deciding, computing and approximating clean answers was also investigated. Clearly, computing clean answers via an explicit and materialized computation of all clean instances is prohibitively expensive and should be avoided whenever possible. Indeed, for a given initial instance \( D \), we could have exponentially many clean instances (in the size of \( D \)).

Answer set programming is a relatively new declarative programming paradigm [Gelfond and Lifschitz 1991; Brewka, Eiter and Truszczynski 2011]. It has been successfully used to implicitly specify in general logical terms, say \( G \), all the solutions of a general, usually combinatorial problem. In this work, we introduce answer set programs (ASPs), in the form of disjunctive Datalog with stable model semantics [Eiter, Gottlob and Mannila 1997], to specify the class \( C(D, \Sigma) \) of clean instances for \( D \) w.r.t. \( \Sigma \). For each instance \( D \) and set \( \Sigma \) of MDs, we show how to build an answer set program \( \Pi(D, \Sigma) \) whose stable models are in one-to-one correspondence with the instances in \( C(D, \Sigma) \).

The cleaning program \( \Pi(D, \Sigma) \) axiomatizes the class \( C(D, \Sigma) \). Hence reasoning from/with the program amounts to reasoning with the full class \( C(D, \Sigma) \). In particular, clean query answers can be obtained from the original instance \( D \) by skeptical (aka. cautious) reasoning on the program.

Answer set programs have been used before in consistent query answering (CQA) [Arenas, Bertossi, and Chomicki 1999; Bertossi 2006; Chomicki 2007; Bertossi 2011], in the form of repair programs, that specify the repairs of a database instance w.r.t. a set of integrity constraints (ICs) [Arenas, Bertossi, and Chomicki 2003; Greco, Greco and Zumpano 2003; Barcelo, Bertossi and Bravo 2003; Eiter et al. 2008; Camiùpan and Bertossi 2010; Francconi et al. 2001]. However, MDs cannot be treated as classical ICs. In particular, the matching functions and the lattice-theoretic structure of the domains, with the induced domination order, create a substantially different scenario, where new challenges arise. Furthermore, the semantics of MDs is quite different from that of classical ICs, and repair techniques for CQA cannot be straightforwardly used for ER via MDs or for clean query answering (cf. [Gardezi, Bertossi and Kirinja 2011] for a discussion).

We statically analyze the cleaning programs, in terms of syntactic structure and complexity. In particular, we show that their expressive power is appropriate for our application in ER, and is in line with the computational complexity of computing clean instances and clean query answers [Bertossi, Kolahi and Lakshmanan 2011]. We also show how to use cleaning programs with the skeptical semantics for the computation of clean answers from the original database, with a data complexity that matches the intrinsic data complexity of clean query answering.

The Swoosh approach to ER was proposed in [Benjelloun et al. 2009], as a generic and procedural specification of ER mechanisms. Special attention receives the common "union-case" of ER, that treats individual records as sets of triples of the form \((id, attr, value)\), i.e. as objects. An ER step basically matches values by producing their union; and the resulting value dominates the original values w.r.t. information contents. In [Bertossi, Kolahi and Lakshmanan 2011], the Swoosh’s union-case was reconstructed in terms of MDs. In this work, we provide, as an additional contribution, a declarative version of the union-case of Swoosh via MDs and their cleaning programs.

This paper is structured as follows. Section 2 introduces the necessary background on relational databases, matching dependencies and their semantics, and disjunctive Datalog programs. In Section 3, we introduce the cleaning programs that specify the clean instances w.r.t. a set of MDs. They are used in Section 4 to do clean query answering. In Section 5 we analyze and transform the cleaning programs, addressing some complexity issues. In Section 6 we present a declarative version of the union case of Swoosh. In Section 7, we obtain a few final conclusions. Many more details in general, full proofs of results, and also examples with the DLV system [Leone et al. 2006], can be found in the extended version of this paper [Bahmani et al. 2011].

2 Preliminaries

We consider relational schemas \( R \) with a possibly infinite data domain \( U \), a finite set of database predicates, e.g. \( R \), and a set of built-in predicates, e.g. \( = \neq \). Each \( R \in R \) has attributes, say \( A_1, \ldots, A_n \), each of them with a domain \( Dom_{A_i} \subseteq U \). We may assume that the \( A_i \)s are different, and different predicates do not share attributes. However, different attributes may share the same domain.

An instance \( D \) for \( R \) is a finite set of ground atoms (or tuples) of the form \( R(c_1, \ldots, c_n) \), with \( R \in R \), \( c_i \in Dom_{A_i} \). We will assume that tuples have identifiers, as in Example 1. They allow us to compare extensions of the same predicate in different instances, and trace changes of attribute values. Tuple identifiers can be accommodated by adding to each predicate \( R \in R \) an extra attribute, \( T \), that acts as a key. Then, tuples take the form \( R(t, c_1, \ldots, c_n) \), with \( t \) a value for \( T \). Most of the time we leave the tuple identifier implicit, or we use it to denote the whole tuple. More precisely, if \( t \) is a tuple identifier in an instance \( D \), then \( t^D \) denotes the entire atom, \( R(\bar{c}) \), identified by \( t \). Similarly, if \( A \) is a list of attributes of predicate \( R \), then \( t^D[A] \) denotes the tuple identified by \( t \), but restricted to the attributes in \( A \). We assume that tuple identifiers are unique across the entire instance.

Schema \( R \) determines a language \( L(R) \) of first-order...
(FO) predicate logic. A conjunctive query is a formula of $L(R)$ of the form $Q(\bar{x}) : \exists y(P_1(x_1) \land \ldots \land P_m(x_m))$, where $P_i \in R$, $\bar{x} = (x_1, \ldots, x_k) \land \bar{y}$ is the list free variables of the query, say $\bar{x} = x_1 \ldots x_k$. An answer to $Q$ in instance $D$ is a sequence $(a_1, \ldots, a_k) \in \mathbb{U}^k$ that makes $Q$ true in $D$, denoted $D \models Q(a_1, \ldots, a_k)$. $Q(D)$ denotes the set of answers to $Q$ in $D$, and can be seen and treated as an instance for an “answer” relational schema, possibly different from the original one.

For a schema $R$ with predicates $R_1[\bar{L}_1], R_2[\bar{L}_2]$, with lists of attributes $\bar{L}_1, \bar{L}_2$ respectively, a matching dependency (MD) [Fan et al. 2009] is an expression of the form
\[
\varphi : \quad R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \rightarrow R_1[\bar{Y}_1] \approx R_2[\bar{Y}_2]. \tag{1}
\]
Here, $\bar{X}_1, \bar{Y}_1$ are sublists of $\bar{L}_1$, and $\bar{X}_2, \bar{Y}_2$ sublists of $\bar{L}_2$. The lists $\bar{X}_1, \bar{X}_2$ (also $\bar{Y}_1, \bar{Y}_2$) are comparable, i.e. the attributes in them, say $X_j^1, X_j^2$, are pairwise comparable in the sense that they share the same data domain $Dom_a$ on which a binary similarity (i.e. reflexive and symmetric) relation $\approx_j$ is defined.

The MD (1) intuitively states that if, for an $R_1$-tuple $t_1$ and an $R_2$-tuple $t_2$ in an instance $D$ the attribute values in $t_1^D[\bar{X}_1]$ are similar to attribute values in $t_2^D[\bar{X}_2]$, then the values $t_1^D[\bar{Y}_1]$ and $t_2^D[\bar{Y}_2]$ have to be made identical. This update results in another instance $D'$, where $t_1'^D[\bar{Y}_1] = t_2'^D[\bar{Y}_2]$ holds. W.l.o.g., we may assume that the list of attributes on the RHS of MDs contain only one conjunct (attribute).

For a set $\Sigma$ of MDs, a pair of instances $(D', D''')$ satisfies $\Sigma$ if whenever $D'$ satisfies the antecedents of the MDs, then $D'''$ satisfies the consequents (taken as equalities). If $(D, D) \not\models \Sigma$, we say that $D$ is “dirty” (w.r.t. $\Sigma$). On the other hand, an instance $D$ is stable if $(D, D) \models \Sigma$ [Fan et al. 2009].

In order to enforce an MD on two tuples, making values of attributes identical, we assume that for each comparable pair of attributes $A_1, A_2$ with domain (in common) $Dom_{A_1}$, there is a binary matching function (MF) $m_A : Dom_A \times Dom_A \rightarrow Dom_A$, such that $m_A(a, a')$ is used to replace two values $a, a' \in Dom_A$ whenever necessary. We expect MFS to be idempotent, commutative, and associative [Bertossi, Kolahi and Lakshmanan 2011; Benjelloun et al. 2009].

The structure $(Dom_A, m_A)$ forms a join semilattice, that is, a partial order with a least upper bound (lub) for every pair of elements. The induced partial order $\preceq_A$ on $Dom_A$ is defined by: $a \preceq_A a'$ whenever $m_A(a, a') = a'$. The lub coincides with $m_A$: $lub_{\preceq_A} \{a, a'\} = m_A(a, a')$. We also assume the existence of the greatest lower bounds, $glb(a, a')$ (cf. [Bertossi, Kolahi and Lakshmanan 2011] for a discussion). In the rest of this work, we will assume by default that similarity relations and MFSs are built-in relations.

A chase-based semantics for data cleaning (or entity resolution) with MDs is given in [Bertossi, Kolahi and Lakshmanan 2011]: starting from an instance $D_0$, we identify pairs of tuples $t_1, t_2$ that satisfy the similarity conditions on the left-hand side of a matching dependency $\varphi$, i.e. $t_1^{D_0}[\bar{X}_1] \approx t_2^{D_0}[\bar{X}_2]$ (but not its RHS), and apply an MF on the values for the right-hand side attribute, $t_1^{D_0}[\bar{A}_1], t_2^{D_0}[\bar{A}_2]$, to make them both equal to $m_A(t_1^{D_0}[\bar{A}_1], t_2^{D_0}[\bar{A}_2])$. We keep doing this on the resulting instance, in a chase-like procedure, until a stable instance is reached.

**Definition 1.** Let $D, D'$ be database instances with the same set of tuple identifiers, $\Sigma$ be a set of MDs, and $\varphi : R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \rightarrow R_1[\bar{Y}_1] \approx R_2[\bar{Y}_2]$ be an MD in $\Sigma$. Let $t_1, t_2$ be an $R_1$-tuple and an $R_2$-tuple identifiers, respectively, in both $D$ and $D'$. Instance $D'$ is the immediate result of enforcing $\varphi$ on $t_1, t_2$ in instance $D$, denoted $(D, D')_{[t_1, t_2]} = \varphi$, if

\[(a) \quad t_1'^D[\bar{X}_1] \approx t_2'^D[\bar{X}_2], \text{ but } t_1'^D[\bar{Y}_1] \not\approx t_2'^D[\bar{Y}_2];
\]
\[(b) \quad t_1'^D[\bar{Y}_1] = t_2'^D[\bar{Y}_2] = m_A(t_1^{D}[\bar{Y}_1], t_2^{D}[\bar{Y}_2]); \text{ and}
\]
\[(c) \quad D', D'' \text{ take the same values on every other tuple and attribute.}
\]

**Definition 2.** For an instance $D_0$ and a set $\Sigma$ of MDs, an instance $D_k$ is $(D_0, \Sigma)$-clean if $D_k$ is stable, and there exists a finite sequence of instances $D_1, \ldots, D_{k-1}$ such that, for every $i \in [1, k], (D_{i-1}, D_i)_{[t_1, t_2]} = \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers $t_1, t_2$.

An instance $D_0$ may have several $(D_0, \Sigma)$-clean instances. $C(D_0, \Sigma)$ denotes the set of clean instances for $D_0$ w.r.t. $\Sigma$.

The domain-level relations $a \preceq_A a'$ can be thought of in terms of relative information contents [Bertossi, Kolahi and Lakshmanan 2011]. They can be lifted to a tuple-level partial order, defined by: $t_1 \preceq t_2$ if $t_1[A] \preceq_A t_2[A]$, for each attribute $A$. This in turn can be lifted to a relation-level partial order: $D_1 \preceq D_2$ iff $\forall t_1 \in D_1 \exists t_2 \in D_2 t_1 \preceq t_2$.

When a tuple $t^D$ in instance $D$ is updated to $t^{D'}$ in instance $D'$ by enforcing an MD and applying an MF, it holds that $t^D \preceq t^{D'}$; and the instances $D$ and $D'$ satisfy: $D \preceq D'$. If instances are all “reduced” by eliminating tuples that are dominated by others, the set of reduced instances with $\varphi$ forms a lattice. That is, we can compute the $glb$ and the lub of every finite set of instances w.r.t. $\preceq$. This is a useful result that allows us to compare sets of query answers w.r.t. $\subseteq$. Indeed, the set of clean answers to a query $Q$ from instance $D$ w.r.t. $\Sigma$ is formally defined by $Clean_{\preceq}(Q) := \{glb(Q(D')) \mid D' \in \mathcal{C}(D, \Sigma)\}$ [Bertossi, Kolahi and Lakshmanan 2011].

We will use logic programs $\Pi$ in Database $\forall, \not\exists$, i.e. disjunctive Datalog programs with weak negation [Gelfond and Lifschitz 1991; Eiter, Gottlob and Mannila 1997], and a finite number of rules of the form

$A_1 \lor \ldots \lor A_n \leftarrow P_1, \ldots, P_m, \not N_1, \ldots, \not N_k,$

where $0 \leq n, m, k$, and $A_i, P_j, N_k$ are (positive) atoms. All the variables in the $A_i, N_k$ appear among those in the $P_j$. The constants in program $\Pi$ form the (finite) Herbrand universe $H$ of the program. The ground version of program $\Pi$, $gr(\Pi)$, is obtained by instantiating the variables in $H$ in all possible ways using values from $H$. The Herbrand base $HB$ of $\Pi$ consists of all the possible atoms obtained by instantiating the predicates in $\Pi$ with constants in $H$.

A subset $M$ of $HB$ is a model of $\Pi$ if it satisfies $gr(\Pi)$, i.e.: For every ground rule $A_1 \lor \ldots \lor A_n \leftarrow P_1, \ldots, P_m$, not $N_1, \ldots, not N_k$ of $gr(\Pi)$, if $\{P_1, \ldots, P_m\} \cap M \neq \emptyset$ and $\{N_1, \ldots, N_k\} \cap M = \emptyset$, then $\{A_1, \ldots, A_n\} \cap M \neq \emptyset$. $M$ is a minimal model of $\Pi$ if it is a model of $\Pi$, and $\Pi$ has no model that is properly contained in $M$. $MM(\Pi)$ denotes the class of minimal models of $\Pi$. Now, for
We start by showing that clean query answering is a non-monotonic process.

Example 2. Consider the instance $D$ and the MD $\phi$:

<table>
<thead>
<tr>
<th>R(D)</th>
<th>name</th>
<th>phone</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>John Doe</td>
<td>(613)7654321</td>
<td>Bank St.</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Alex Smith</td>
<td>(514)1234567</td>
<td>10 Oak St.</td>
</tr>
</tbody>
</table>

$\phi$: $R[\text{phone, addr}] \triangleright R[\text{phone, addr}] \Rightarrow R[\text{addr}] \not\equiv R[\text{addr}]$.

$D$ is a stable, clean instance w.r.t. $\phi$. Now consider the query $Q(z): \exists y R(\text{John Doe}, y, z)$, asking for the address of John Doe. In this case, $\text{Clean}^D[\phi](Q) = Q(D) = \{\{\text{Bank St.}\}\}$. 

Now, suppose that $D$ is updated into $D'$:

<table>
<thead>
<tr>
<th>R(D')</th>
<th>name</th>
<th>phone</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>John Doe</td>
<td>(613)7654321</td>
<td>Bank St.</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Alex Smith</td>
<td>(514)1234567</td>
<td>10 Oak St.</td>
</tr>
<tr>
<td>$t_3$</td>
<td>J.Doe</td>
<td>7654321</td>
<td>25 Bank St.</td>
</tr>
</tbody>
</table>

Assuming that "(613)7654321" is "7654321", Bank St. is 25 Bank St., and also $m_{\text{addr}}(\text{Bank St.}, 25 \text{ Bank St.}) = 25 \text{ Bank St.}$, then $D''$ below is the only clean instance:

<table>
<thead>
<tr>
<th>R(D'')</th>
<th>name</th>
<th>phone</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>John Doe</td>
<td>(613)7654321</td>
<td>25 Bank St.</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Alex Smith</td>
<td>6131234567</td>
<td>10 Oak St.</td>
</tr>
<tr>
<td>$t_3$</td>
<td>J.Doe</td>
<td>7654321</td>
<td>25 Bank St.</td>
</tr>
</tbody>
</table>

Now, $\text{Clean}^{D'}[\phi](Q) = Q(D'') = \{\{25 \text{ Bank St.}\}\}$. Clearly, $Q(D) \not\subseteq Q(D')$, even though $D \subseteq D'$.

This example shows that a non-monotonic logical formalism is required to capture the clean instances as its models. Intuitively, when an MD is enforced on two tuples of an instance in a single step of the chase procedure, the tuples are updated to newer versions. The older versions of the tuples should no longer be available during the rest of the chase.

We use answer set programs as the basic formalism to capture this non-monotone chase procedure. More precisely, given an instance $D_0$ and a set $\Sigma$ of MDs, we propose a logic program $\Pi(D_0, \Sigma)$ whose stable models correspond to the $(D_0, \Sigma)$-clean instances.

Program $\Pi(D_0, \Sigma)$ should have rules to "implicitly simulate" a chase sequence, i.e. rules that enforce MDs on pairs of tuples that satisfy certain similarities, create newer versions of those tuples by applying matching functions, and make the older versions of the tuples unavailable for other rules. The idea is to have the stable models of the program correspond to valid chase sequences leading to clean instances.

$\Pi(D_0, \Sigma)$ explicitly eliminates, using program constraints, instances that are the result of illegal applications of MDs. A set of matching applications is illegal if we cannot put them in a chronological order to represent the steps of a chase. That is, there are some matchings that use old versions of tuples that have been replaced by new versions.

To ensure that the matchings are enforced according to an order that correctly represents a chase, we will record pairs of matchings in an auxiliary relation, $\text{Prec}$, in the cleaning program, and explicitly impose an order on $\text{Prec}$ via program constraints.

### 3.1 Cleaning programs for MDs

Let $D_0$ be a given, possibly dirty initial instance w.r.t. a set $\Sigma$ of MDs. The cleaning program, $\Pi(D_0, \Sigma)$, that we will introduce here, contains an $(n + 1)$-ary predicate $R_i$, for each $n$-ary database predicate $R_i$. It will be used in the form $R_i(T, Z)$, where $T$ is a variable for the tuple identifier attribute, and $Z$ is a list of variables standing for the (ordinary) attribute values of $R_i$.

For every attribute $A$ in the schema, with domain $\text{Dom}_A$, the built-in ternary predicate $M_{\bar{A}}$ represents the MF $m_{\bar{A}}$, i.e. $M_{\bar{A}}(a, a', a'')$ means $m_{\bar{A}}(a, a') = a''$. $X \preceq A$ is used as an abbreviation for $M_{\bar{A}}(X, Y, Y)$. When an attribute $A$ (or its domain) does not have a matching function, because it is not affected by an MD, then $\preceq A$ becomes the equality, $=\bar{A}$. For lists of variables $Z_1 = (Z_1^1, \ldots, Z_1^n)$ and $Z_2 = (Z_2^1, \ldots, Z_2^n)$, $Z_1 \preceq Z_2$ denotes the conjunction $Z_1^1 \preceq A_1, Z_2^1 \preceq A_1, \ldots, Z_1^n \preceq A_n, Z_2^n$. Moreover, for each attribute $A$, there is a built-in binary predicate $\preceq A$. For two lists of variables $X_1 = (X_1^1, \ldots, X_1^n)$ and $X_2 = (X_2^1, \ldots, X_2^n)$ representing comparable attribute values, $X_1 \preceq X_2$ denotes the conjunction $X_1^1 \preceq X_2^1, X_1^2 \preceq X_2^2, \ldots, X_1^n \preceq X_2^n$.

The program $\Pi(D_0, \Sigma)$ contains the rules in 1.-7. below:

1. For every tuple (id) $t_{D_0} = R_i(\bar{a})$, the fact $R_i'^2(t, \bar{a})$.
2. For each MD $\phi_i: R_1[X_1] \approx R_2[X_2] \rightarrow R_1[A_1] \equiv R_2[A_2]$, the program rules:
   
   Match$_{\phi_i}(T_1, X_1, Y_1, T_2, X_2, Y_2) \rightarrow \\
   \text{NotMatch}_i(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow \\
   \text{OldVersion}_R(T_1, Z_1) \leftarrow R_i(T_1, Z_1), \text{OldVersion}_R(T_2, Z_2), \\
   \bar{Z}_1 \not\preceq \bar{Z}_2, \bar{Z}_1 \not\preceq \bar{Z}_2'. \quad (i = 1, 2) \\
   \leftarrow \text{NotMatch}_i(T_1, Z_1, T_2, Z_2), \\
   \text{not OldVersion}_R(T_1, Z_1), \text{not OldVersion}_R(T_2, Z_2).

In these rules, the $\tilde{X}_i$s are lists of variables corresponding to lists of attributes on the LHS of the MD, whereas the $Y_i$s are single variables corresponding to the attribute on the RHS of the MD. Also, the $\tilde{Z}_i$s are lists of variables corresponding to all attributes in a tuple. We use this notation to make the association with attributes or lists thereof in MDs easier.

These rules are used to enforce the MDs whenever the necessary similarities hold for two tuples. The first rule in 2. specifies that in that case, a matching may or may not take place, but the latter is acceptable only if one of the involved tuples is used for another matching, and replaced by a newer version. This is enforced using the third rule, actually a program constraint, that has the effect of filtering out stable models where the conjunction in its body becomes true.
More precisely, predicate $\text{OldVersion}_{Rt}$ contains different versions of every tuple (id) in relation $Rt$ which has been replaced by a newer version (during the ER process). For each tuple identifier $t$ there could be many atoms of the form $Rt′(t, a)$ corresponding to different versions of the tuple associated with $t$ that represent the evolution of the tuple during the enforcement of MDs. The second rule specifies when an atom $Rt′(t, a)$ for a tuple identifier $t$ has been replaced by a newer version $Rt′(t, a′)$, with $a \preceq a′$, due to a matching.

The program constraint in 2. above states that if: (a) we have “live”, never replaced versions of two tuples $t_1$ and $t_2$ from relations $R_1$ and $R_2$, respectively, (b) the similarity conditions holds for them according to an MD, and (c) both are not matched (together or with some other tuples), then the model should be rejected. That is, $t_1$ and $t_2$ have to be either matched together, or be replaced by newer versions (becoming unavailable). This constraint enforces at least one match for a tuple that satisfies some match condition.

For convenience, below we refer to the various atoms associated with a given tuple identifier $t$ as versions of the tuple identifier $t$.

When the two predicates appearing in $\phi_j$ are the same, say $R_1$, the first rule becomes symmetric w.r.t. every two atoms $R_1′(t_1, a_1)$ and $R_1′(t_2, a_2)$ that satisfy the body of the rule. We need to make sure that if the matching takes place for these two tuples, then both $\text{Match}_{\phi_j}(t_1, a_1, t_2, a_2)$ and $\text{Match}_{\phi_j}(t_2, a_2, t_1, a_1)$ exist. Thus, for every such MD, we need a rule of the following form

$\text{Match}_{\phi_j}(T_2, X_2, Y_2, T_1, X_1, Y_1) \leftarrow \text{Match}_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2)$.

3. Rules to insert new tuples into $R_1, R_2$, as a result of enforcing $\phi_j$ ($M_j$ stands for the MF for the RHS of $\phi_j$):

$R_1′(T_1, X_1, Y_1) \leftarrow \text{Match}_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_j(Y_1, Y_2, Y_3)$.

$R_2′(T_2, X_2, Y_2) \leftarrow \text{Match}_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_j(Y_1, Y_2, Y_3)$.

4. For every two matchings applicable to different versions of a tuple with a given identifier, we record in $\text{Prec}$ the relative order of the matchings. The matching applied to the smaller version of the tuple w.r.t. $\preceq$ has to precede the other.

$\text{Prec}(T_1, Z_1, T_2, Z_2, Z_1′, T_3, Z_3) \leftarrow \text{Match}_{\phi_j}(T_1, Z_1, T_2, Z_2), \text{Match}_{\phi_j}(T_1, Z_1′, T_3, Z_3), Z_1 \preceq Z_1′, Z_i \neq Z_i′$.

We need similar rules (four in total) for the cases where the common tuple identifier variable $T_1$ appears in different components of the two $\text{Match}$ predicates.

5. Each version of a tuple identifier can participate in more than one matching only if at most one of them changes the tuple. For every two matchings applicable to a single version of a tuple identifier, we record in $\text{Prec}$ the relative order of the two matchings. The matching that produces a new version for the tuple has to come after the other matching. If both of the matchings do not produce a new version of the tuple, they can be applied in any order, making unnecessary to record their relative order in $\text{Prec}$.

$\text{Prec}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \leftarrow \text{Match}_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), \text{Match}_{\phi_j}(T_1, X_1, Y_1, T_3, X_3, Y_3), M_k(Y_1, Y_3, Y_4), Y_1 \neq Y_4$.

This rule says that, in case (a ground version of) a tuple $(T_1, X_1, Y_1)$ participates in two matching, via MDs $\phi_j$ and $\phi_k$, the tuple changes according to $\phi_k$ as captured by the last two body atoms that use $\phi_k$’s matching function $M_k$, then the matching via $\phi_k$ must come after the matching via $\phi_j$. By this same rule, the reverse $\text{Prec}$-order could also be true, but we will disallow having both by imposing conditions on $\text{Prec}$, making it a partial order (see below). By the first rule(s) in 2. above, a stable model can always choose between doing a matching or not, and then choosing between one of the two possible $\text{Prec}$-orders.

As in rule 4. above, we need four rules of this form, for different possible appearances of the common variable $T_1$. This rule disallows two matchings that produce incomparable versions of a tuple w.r.t. $\preceq$, because $\text{Prec}$ is antisymmetric (due to rules 6. below). As a consequence, every two matchings applicable to a given tuple identifier will fire one of the two rules 4. or 5., and they will have a relative order recorded in $\text{Prec}$, unless they both do not change the tuple.

6. Rules for making $\text{Prec}$ a reflexive, antisymmetric and transitive relation, respectively:

$\text{Prec}(T_1, Z_1, T_2, Z_2, T_1, Z_1′, T_3, Z_3) \leftarrow \text{Match}_{\phi_j}(T_1, Z_1, T_2, Z_2)$.

$\text{Prec}(T_1, Z_1, T_2, Z_2, T_1, Z_1′, T_3, Z_3), \text{Prec}(T_1, Z_1′, T_3, Z_3, T_1, Z_1, T_2, Z_2), (T_1, Z_1′, T_3, Z_3) \neq (T_1, Z_1, T_3, Z_3)$.

$\text{Prec}(T_1, Z_1, T_2, Z_2, T_1, Z_1′, T_3, Z_3), \text{Prec}(T_1, Z_1′, T_3, Z_3, T_1, Z_1, T_2, Z_2), (T_1, Z_1, T_2, Z_2) \neq (T_1, Z_1′, T_3, Z_3)$.

Notice that we do not use $\text{Prec}$ in body conditions. In consequence, the main rules around it are the last two program constraints. They are used to to eliminate instances (models) that result from illegal applications of MDs.

7. Finally, rules to collect in $Rt′$ the latest version of each tuple for every predicate $Rt$; they are used to form the clean instances.

$\text{Rt′}(T_1, Z_1) \leftarrow \text{Rt′}(T_1, Z_1), \text{not OldVersion}_{Rt′}(T_1, Z_1)$.

Notice that the rules in 2. above are the only ones that depend on an essential manner on the particular MDs at hand. Rules 1. are just the facts that represent the initial, underlying database. All the other rules are basically generic, and could be used by any cleaning program, as long as there is a correspondence between the predicates $\text{Match}_\varphi$ with the MDs $\varphi$, for which the former have subindices for the latter.

Notice that, given a relational schema and a set of MDs on it, a program like the one above can be automatically created, and can be used for that schema and MDs. Only the facts of the program depend on the actual relational instance at hand. An alternative to our approach would be to build a single program that can be used with any schema and finite
Example 3. Consider relation $R(A, B)$, and set $\Sigma$ with:

$$\phi_1 : R[A] \approx R[A] \rightarrow R[B] \approx R[B],$$

$$\phi_2 : R[B] \approx R[B] \rightarrow R[A] \approx R[A].$$

Assume that exactly the following similarities hold: $a_1 \approx a_2, b_2 \approx b_3$; and the MFs are as follows:

$$M_B(b_1, b_1),$$

$$M_B(b_3, b_3),$$

$$M_B(b_1, b_23, b_123).$$

Enforcing $\Sigma$ on $D_0$ according to Definition 2 results in two chase sequences, each enforcing the MDs in a different order, and two final stable clean instances $D_1$ and $D_2$.

<table>
<thead>
<tr>
<th>$D_0$</th>
<th>$A$</th>
<th>$B$</th>
<th>$D_1$</th>
<th>$A$</th>
<th>$B$</th>
<th>$D_2$</th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$t_1$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$t_1$</td>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$t_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$t_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$a_3$</td>
<td>$b_3$</td>
<td>$t_3$</td>
<td>$a_3$</td>
<td>$b_3$</td>
<td>$t_3$</td>
<td>$a_3$</td>
<td>$b_3$</td>
</tr>
</tbody>
</table>

The cleaning program $\Pi(D_0, \Sigma)$ has the following rules (skipping rules 6).

1. $R'(t_1, a_1, b_1) \iff R'(t_2, a_2, b_2) \iff R'(t_3, a_3, b_3)\).

2. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2) \quad \land \quad \text{NotMatch}(T_1, X_1, Y_1, T_2, X_2, Y_2) \iff R'(T_1, X_1, Y_1) \iff R'(T_2, X_2, Y_2), X_1 \approx X_2, Y_1 \neq Y_2.$

3. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2) \quad \lor \quad \text{NotMatch}(T_1, X_1, Y_1, T_2, X_2, Y_2) \iff R'(T_1, X_1, Y_1) \iff R'(T_2, X_2, Y_2), Y_1 \approx Y_2, Y_1 \neq Y_2.$

4. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2) \quad \iff \quad \text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2), \quad \text{M}_B(Y_1, Y_2, Y_3).$

5. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_3, X_3, Y_3) \iff \text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2), \quad \text{M}_B(Y_1, Y_2, Y_3).$

6. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_3, X_3, Y_3) \iff \text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2), \quad \text{M}_B(Y_1, Y_2, Y_3).$

7. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_3, X_3, Y_3) \iff \text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2), \quad \text{M}_B(Y_1, Y_2, Y_3).$

8. $\text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_3, X_3, Y_3) \iff \text{Match}(T_1, X_1, Y_1, T_2, X_2, Y_2), \quad \text{M}_B(Y_1, Y_2, Y_3).$

From them we can read off the two clean instances $D_1, D_2$ for $D_0$ that were obtained from the chase. The stable models of the program can be computed using the DLV system [Leone et al. 2006]. The DLV code for this example can be found in [Bahmani et al. 2011, appendix B].

Theorem 1. There is a one-to-one correspondence between $C(D_0, \Sigma)$ and the set $SM(\Pi(D_0, \Sigma))$ of stable models of the cleaning program $\Pi(D_0, \Sigma)$. More precisely, the clean instances for $D_0$ w.r.t. $\Sigma$ are exactly the restrictions of the elements of $SM(\Pi(D_0, \Sigma))$ to schema $R^c$.

The restriction of the stable models to the relational schema $R^c$ is due to the fact that they also have extensions for the auxiliary predicates used in the programs.

Proof sketch. The proof of the theorem consists of two parts.

1. We need to show that for every $(D_0, \Sigma)$-clean instance $D_k$ (obtained through a chase as in Definition 2), we can construct a set of atoms $S_{D_k}$ that is a stable model for the logic program $\Pi(D_0, \Sigma)$. Let $D_1, \ldots, D_{k-1}$ be the instances in the chase sequence that leads to clean instance $D_k$. The tuples in these instances will give us different versions of each tuple identifier that come to existence due to the matchings, and hence we can populate predicates $R^c_i, \text{OldVersion}_{R^c_i}$, and $R^c_i$ of the logic program. The MDs enforced in this chase, the relative order of enforcing them, and the tuples involved in each MD application will let us populate the predicates $\text{Match}_{\phi_i}, \text{NotMatch}_{\phi_i}$, and $\text{Prec}$. Then, we show that the set $S_{D_k}$ consisting of all these ground atoms satisfies the program and it is a minimal model.

2. We need to show that, for every stable model $S$ of the program $\Pi(D_0, \Sigma)$, we can construct a $(D_0, \Sigma)$-clean instance $S$. To construct $S$, we let $\delta_{\phi_i} = \bar{a}$ for every relation $R_i$ and every tuple identifier $t$ of relation $R_i$ such that $R_i^c(t, \bar{a}) \in S$. Then using the $\text{Match}_{\phi_i}$ atoms in $S$ and also the relative order of these matchings in $\text{Prec}$, we identify which MDs are applied to which tuples and in what order; and therefore we can construct the sequence of instances $D_1, \ldots, D_k$ that reflect enforcing these MDs in the same order imposed by $\text{Prec}$. Then, we show that this is actually a valid chase sequence, and the resulting instance $D_k$ is a stable instance that is equal to $S$.

4. Query Answering

We can use the cleaning program $\Pi(D_0, \Sigma)$ to compute the clean answers to a query $Q$ posed to $D_0$, as defined in Section 2. The clean answers were defined by taking into account the underlying lattices, as the $\text{glb}$ of all the sets of
answers that can be obtained by evaluating the query on a clean instance. This is not the same as certain (or skeptical) answers, i.e., the set-theoretic intersection of all the answers from every clean instance, and therefore it is not equivalent to skeptical answer of the logic program. In this section we provide a mechanism for computing clean answers while still using skeptical query answering.

Given an FO query \( Q(x_1, \ldots, x_n) \), with free variables standing for attributes \( A_1, \ldots, A_n \) of \( \mathcal{R} \), and defined by a formula \( \varphi(\bar{x}) \), (with \( \bar{x} = x_1, \ldots, x_n \)), a non-disjunctive and stratified query program \( \Pi(Q) \) can be obtained from \( \varphi \), using a standard transformation [Lloyd 1987]. It contains an answer predicate \( \text{Ans}_Q(\bar{x}) \) to collect the answers to \( Q \), and rules defining it, of the form \( \text{Ans}_Q(\bar{x}) \leftarrow B(\bar{x}') \), where the \( B \)s are conjunctions of literals (i.e., atoms or negations not \( A \) thereof). The \( R \)-atoms in \( Q \), with \( R \in \mathcal{R} \), are replaced in \( \Pi(Q) \) by \( R^* \)-atoms.

We can obtain reduced answer sets (cf. Section 2) by adding two new rules to \( \Pi(Q) \):\footnote{Notice that \( \preceq \) in the second rule is defined in terms of the built-in relations \( \mathcal{M}_A \).}

\[
\text{Ans}_Q(\bar{x}) \leftarrow \text{Ans}_Q(\bar{y}), \text{not Dominated}_Q(\bar{x}).
\]

\[
\text{Dominated}_Q(\bar{x}) \leftarrow \text{Dominated}_Q(\bar{y}), \bar{x} \preceq \bar{y}, \bar{x} \neq \bar{y}.
\]

The stable models \( \mathcal{S} \) of \( \Pi(D_0, \Sigma, Q) := \Pi(D_0, \Sigma) \cup \Pi(Q) \) are the stable models of \( \Pi(D_0, \Sigma) \) expanded with extensions \( \text{Ans}_Q^\preceq(\mathcal{S}) \) for predicate \( \text{Ans}_Q^\preceq \). Those extensions, as database instances, are already reduced. Assume that \( \text{SM}(\Pi(D_0, \Sigma, Q)) = \{S_1, \ldots, S_m\} \). It holds that \( \text{Clean}_Q^\preceq(D_0)(\mathcal{S}) = \text{glb}_Q(\{\text{Ans}_Q^\preceq(S_i) \mid i = 1, \ldots, m\}) = \text{Red}_Q(\{\text{glb}_Q(\{\bar{a}_1, \ldots, \bar{a}_m\} \mid \bar{a}_i \in \text{Ans}_Q^\preceq(S_i), i = 1, \ldots, m\}) \), where \( \text{Red}_Q \) produces the reduced version of a set.

Now we show how the program \( \Pi(D_0, \Sigma, Q) \) can be modified, so that the clean answers to query \( Q \) can be obtained by running the program under the skeptical semantics. Given \( \text{Ans}_Q^\preceq(S_i) \), i.e., the set of answers to \( Q \) from the clean instance corresponding to the stable model \( S_i \), we define its downward expansion by:

\[
\text{Ans}_Q^\preceq^\downarrow(S_i) := \{\bar{b} \mid \bar{b} \preceq \bar{a}, \text{ for some } \bar{a} \in \text{Ans}_Q^\preceq(S_i)\}.
\]

\( \text{Ans}_Q^\preceq(S_i) \) contains all the atoms in \( \text{Ans}_Q^\preceq(S_i) \) and everything below w.r.t. the \( \preceq \) lattice. Since \( \text{Ans}_Q^\preceq(S_i) \) is finite, \( \text{Ans}_Q^\preceq^\downarrow(S_i) \) is also finite, because we consider finite lattices.

**Proposition 1.** Let \( D_0 \) be an instance, \( \Sigma \) be a set of MDs, and \( Q \) be a query. Let \( \text{SM}(\Pi(D_0, \Sigma, Q)) = \{S_1, \ldots, S_m\} \), then \( \text{glb}_Q(\{\text{Ans}_Q^\preceq(S_i) \mid i = 1, \ldots, m\}) = \text{Red}_Q(\{\text{Ans}_Q^\preceq(S_i) \mid i = 1, \ldots, m\}) \).

As a consequence of this result, the clean answers can be obtained by taking the (set-theoretic) intersection of all sets \( \text{Ans}_Q^\preceq(S_i) \) followed by a final reduction instead of taking the \( \text{glb} \) over all sets \( \text{Ans}_Q^\preceq(S_i) \). This can be achieved directly through \( \Pi(D_0, \Sigma, Q) \) by adding to it the following rule:

\[
\text{Ans}_Q^\preceq(\bar{y}) \leftarrow \text{Ans}_Q^\preceq(\bar{x}), \bar{y} \preceq \bar{x}, \text{Dom}_L(\bar{y}).
\]

Here, \( \text{Dom}_L(\cdot) \) is stands for the cartesian product of the finite domains \( \text{Dom}_A \) for the local lattices \( \mathcal{L}_A \).

The new rule will expand each stable model by adding finitely many \( \text{Ans}_Q^\preceq^\downarrow(\bar{b}) \) atoms for every \( \text{Ans}_Q^\preceq(\bar{a}) \) atom, where \( \bar{b} \preceq \bar{a} \). The values for \( \bar{y} \) are taken from \( \text{Dom}_L \). Then each stable model will contain the atoms in the \( \text{glb} \) of all stable models, restricted to the \( \text{Ans}_Q^\preceq(S_i) \) predicate, and therefore the intersection of all stable models will contain the \( \text{glb} \). In this way we can obtain the clean answers to query \( Q \).

We have just described a way to compute, by means of the downward expanded programs, the clean answers to a query \( Q \). In this way we avoid a separate and off-line gathering of query answers from each of the stable models for later combination via the \( \text{glb} \). The manifold programs (MF) [Faber and Woltran 2011] offer another alternative for using a single ASP for the whole task. Here we will just sketch the way they can be used in this direction, actually in combination with an extension of ASP with sets and unions thereof [Calimeri et al. 2009]. More details on this extension will be given in Section 6.1.

Given a program \( \Pi \), an MF program for \( \Pi \), say \( \text{MF}(\Pi) \), extends \( \Pi \) by collecting brave or skeptical atomic consequences from what would have been \( \Pi \)-now a part of \( \text{MF}(\Pi) \)-and using them for further processing by \( \text{MF}(\Pi) \).

In our case, properly marked brave consequences from \( \Pi(D_0, \Sigma, Q) \) of the form \( \text{Ans}_Q^\preceq(\bar{a}) \), with \( \bar{a} \in \text{SM}(\Pi(D_0, \Sigma, Q)) \), can be further used by \( \text{MF}(\Pi(D_0, \Sigma, Q)) \) to compute the \( \text{glb} \)s. For this, \( \text{MF}(\Pi(D_0, \Sigma, Q)) \) includes rules of the form (we give a high-level description of them):

\[
\text{glb}_Q(\bar{x}, U) \leftarrow U = \# \text{Union}(\{\bar{y}, U\}), \text{glb}_Q(\bar{u}, U'), \bar{x} = \text{glb}_Q(\bar{u}, \bar{y}).
\]

\[
\text{glb}_Q(\bar{x}, \{\bar{y}\}) \leftarrow \text{Dom}(\bar{x}).
\]

\[
\text{PAns}_Q(\bar{x}) \leftarrow \text{glb}_Q(\bar{x}, \{\bar{x}_1, \ldots, \bar{x}_m\}), \text{Ans}_Q(\bar{x}_1)^{S_1}, \ldots, \text{Ans}_Q(\bar{x}_m)^{S_m}.
\]

\[
\text{Dominated}_Q^\downarrow(\bar{x}) \leftarrow \text{PAns}_Q(\bar{y}), \bar{x} \preceq \bar{y}, \bar{x} \neq \bar{y}.
\]

\[
\text{CAns}(\bar{x}) \leftarrow \text{PAns}_Q(\bar{x}), \text{not Dominated}_Q^\downarrow(\bar{x}).
\]

Here, \( \text{glb}_Q(\bar{x}, U) \) is a binary predicate that says that tuple \( \bar{x} \) is the \( \text{glb}_Q \) of set \( U \); and is defined by recursion and associativity: \( \text{glb}_Q(\{\bar{y}\} \cup U) = \text{glb}_Q(\bar{y}, \text{glb}_Q(U')) \). \( \text{glb}_Q(\bar{u}, \bar{y}) \) is a function that produces the \( \text{glb}_Q \) of two tuples. The first two rules use the extension of ASP with sets and operations with them (as in Section 6.1). They recursively compute the \( \text{glb} \) of a set. The domain predicate, \( \text{Dom} \), is associated to the cartesian product of the finite attribute domains involved.

The third rule computes the pre-answers by combination into the \( \text{glb} \) of brave answers obtained from the \( \text{Ans}_Q(\bar{x}_1)^{S_1} \). The next rule computes the dominated answers. The last one computes clean answers by discarding pre-answers that are dominated by other pre-answers. Notice that the “manifold part” of the program above is used to form a set of values for a higher-level aggregation. The sets and the aggregation do no appear in the properly manifold part.

**5 Analysis of Cleaning Programs**

In this section we investigate the properties of the cleaning programs in terms of their syntactic structure, and by doing so, shedding some light of their expressive power and computational complexity. At the same time, this analysis will provide upper-bounds for natural computational problems in
relation to entity resolution via MDs. In this direction, we first review the main classes of Datalog programs, and some known complexity results for them.

With Datalog\(_{\not s}\), we denote the subclass of programs in Datalog\(_{\not s}\) that have stratified negation [Eiter and Gottlob 1995]. If a program is stratified, then its stable models can be computed bottom-up by propagating data upwards from the underlying extensional database (that corresponds to the set of facts of the program), and making sure to minimize the selection of true atoms from the disjunctive heads. Since the latter introduces a form of non-determinism, a program may have several stable models. If the program is non-disjunctive, there is a single stable model, and it can be computed in polynomial time in the size of the extensional database.

Datalog\(_{\not s}\) extends the classes Datalog, Datalog\(_{\not s}\), and Datalog\(_{\not s}\) of non-disjunctive, classical Datalog programs, Datalog programs with stratified negation, and Datalog programs with negation, resp. [Abiteboul, Hull, and Vianu 1995; Ceri, Gottlob, and Tanca 1989]. Datalog\(_{\not s}\) extends Datalog\(_{\not s}\). Programs in Datalog and Datalog\(_{\not s}\) have a single stable model that can be computed in a bottom-up manner starting from the extensional database (EDB), i.e. from the set of facts. In general, disjunctive Datalog programs and those in Datalog\(_{\not s}\) (without stratified negation) may have multiple stable models.

The (likely) higher expressive power of Datalog\(_{\not s}\) w.r.t. Datalog and Datalog\(_{\not s}\) is reflected in, or caused by, the (probable) difference in computational complexity. The problem of deciding if a ground atom A is entailed by a program \(\Pi \in \text{Datalog}_{\not s}\), i.e. if A is true in all the stable models of \(\Pi\), is \(\Pi_{2}^{\text{-complete}}\) in the size of the EDB. This decision problem is also referred to as skeptical (cautious) query answering. The same problem can be solved in polynomial time for programs in Datalog and Datalog\(_{\not s}\) (cf. [Dantsin et al. 2001] for more details).

**Proposition 2.** The cleaning programs \(\Pi(D, \Sigma)\) belong to the class Datalog\(_{\not s}\).

As a consequence of this result, the stable models of the programs introduced in Section 3.1 can be obtained with a bottom-up computation, which is in line with the chase procedure of Definition 2, that defines the clean instances.

It is worth noticing that the data complexity of skeptical query evaluation for programs in Datalog\(_{\not s}\) is the same as for programs with unstratified negation, i.e. for the class Datalog\(_{\not s}\), i.e. \(\Pi_{2}^{\text{-complete}}\) [Eiter and Gottlob 1995; Dantsin et al. 2001; Gelfond and Leone 2002].

However, computing in polynomial time in the size of the extensional database.

Datalog\(_{\not s}\) extends the classes Datalog, Datalog\(_{\not s}\), and Datalog\(_{\not s}\) of non-disjunctive, classical Datalog programs, Datalog programs with stratified negation, and Datalog programs with negation, resp. [Abiteboul, Hull, and Vianu 1995; Ceri, Gottlob, and Tanca 1989]. Datalog\(_{\not s}\) extends Datalog\(_{\not s}\). Programs in Datalog and Datalog\(_{\not s}\) have a single stable model that can be computed in a bottom-up manner starting from the extensional database (EDB), i.e. from the set of facts. In general, disjunctive Datalog programs and those in Datalog\(_{\not s}\) (without stratified negation) may have multiple stable models.

The (likely) higher expressive power of Datalog\(_{\not s}\) w.r.t. Datalog and Datalog\(_{\not s}\) is reflected in, or caused by, the (probable) difference in computational complexity. The problem of deciding if a ground atom A is entailed by a program \(\Pi \in \text{Datalog}_{\not s}\), i.e. if A is true in all the stable models of \(\Pi\), is \(\Pi_{2}^{\text{-complete}}\) in the size of the EDB. This decision problem is also referred to as skeptical (cautious) query answering. The same problem can be solved in polynomial time for programs in Datalog and Datalog\(_{\not s}\) (cf. [Dantsin et al. 2001] for more details).

**Proposition 2.** The cleaning programs \(\Pi(D, \Sigma)\) belong to the class Datalog\(_{\not s}\).

As a consequence of this result, the stable models of the programs introduced in Section 3.1 can be obtained with a bottom-up computation, which is in line with the chase procedure of Definition 2, that defines the clean instances.

It is worth noticing that the data complexity of skeptical query evaluation for programs in Datalog\(_{\not s}\) is the same as for programs with unstratified negation, i.e. for the class Datalog\(_{\not s}\), i.e. \(\Pi_{2}^{\text{-complete}}\) [Eiter and Gottlob 1995; Dantsin et al. 2001; Gelfond and Leone 2002].

Repair programs for CQA under ICs, also belong to the class Datalog\(_{\not s}\) [Cantinptr and Bertossi 2010]; and their relatively high expressive power is really needed to specify database repairs, because the intrinsic data complexity of CQA is provably \(\Pi_{2}^{\text{-complete}}\) (cf. [Bertossi 2011] for a survey of complexity results in CQA). In the case of cleaning programs two natural questions arise. First, whether they provide an expressive power that exceeds the one needed for clean query answering. Secondly, whether we can obtain an informative upper bound on the complexity of clean query answering.

These questions are closely related to the properties of the cleaning programs as determined by their syntactic structure. Actually, it turns out that their syntactic structure can be simplified. More precisely, a cleaning program can be transformed into one that is non-disjunctive. To undertake this task, we need some terminology.

Let \(\Pi \in \text{Datalog}_{\not s}\) and \(\Pi^{0}\) be its ground version. The dependency graph, \(DG(\Pi^{0})\), is a directed graph whose nodes are literals of \(\Pi^{0}\). There is an arc from \(L_{1}\) to \(L_{2}\) iff there is a rule in \(\Pi^{0}\) where \(L_{1}\) appears positive in the body and \(L_{2}\) appears in the head. \(\Pi\) is head-cycle free (HCF) iff \(DG(\Pi^{0})\) has no cycle through two literals that belong to the head of a same rule [Ben-Eliyahu and Dechter 1994].

HCF programs in Datalog\(_{\not s}\) can be transformed into equivalent non-disjunctive programs, i.e. with the same stable models [Ben-Eliyahu and Dechter 1994]. That is, they can be written as programs in Datalog\(_{\not s}\).

**Proposition 3.** Every cleaning program \(\Pi(D_{0}, \Sigma)\) is HCF, and hence can be transformed into an equivalent non-disjunctive program in Datalog\(_{\not s}\).

The transformation is standard. Each disjunctive rule generates as many non-disjunctive rules as atoms in the head, by keeping one at a time in the head, and moving the others in negated form to the body.

In general, for a HCF program, checking if a set of atoms is a stable model can be done in polynomial time [Gelfond and Leone 2002]. However, checking if a set of atoms is contained in a stable model becomes an \(NP\)-complete problem [Ben-Eliyahu and Dechter 1994]. In our case, checking if an instance \(D'\) is a clean instance (for \(D_{0}\) and \(\Sigma\)), amounts to checking if \(\bar{D'}\) is contained in stable model of \(\Pi(D_{0}, \Sigma)\), since the stable models also contain atoms other than \(R\)-atoms. Equivalently, \(\bar{D'}\) does not contain the “cleaning-history” (chase steps) as represented by those other atoms in a stable model. That cleaning-history seems to be necessary to check if \(\bar{D'}\) is a clean instance (just checking stability, i.e. if \(\bar{D'}, \bar{D''}\vdash \Sigma\) is the easy part). In consequence, directly from Proposition 3 we can only obtain that checking if an instance is a clean instance belongs to \(NP\).

The data complexity of skeptical query answering from program in Datalog\(_{\not s}\) is \(co-NP\)-complete [Dantsin et al. 2001]. In consequence, the decision problem of skeptical query answering from \(\Pi(D_{0}, \Sigma)\) belongs to the class \(co-NP\). From this result and Theorem 1, we obtain

**Proposition 4.** For a set \(\Sigma\) of MDs, and a FO query \(Q(\bar{x})\), deciding if a tuple \(\bar{c}\) is a clean answer to \(Q\) from an instance \(D_{0}\) belongs to the class \(co-NP\) (in the size of \(D_{0}\)).

This result should be contrasted with the \(co-NP\)-complete data complexity of deciding clean query answers presented in [Bertossi, Kolahi and Lakshmanan 2011, Theorem 3]. We have reobtained the membership of \(co-NP\) via cleaning programs, but, more importantly, we can conclude that our cleaning programs are not overkilling the problem of clean query answering, and that we need all the expressive power that they provide.

\(\text{To be precise, we have to use program }\Pi(D_{0}, \Sigma, Q)\text{ expanded with rule (2), which actually adds to }D_{0}\text{ the extension of }\text{Dom}_C\text{. However, the latter could be left as a fixed parameter.}\)
The proof of co-NP-hardness for clean query answering in [Bertossi, Kolahi and Lakshmanan 2011] can be easily modified to prove that certain query answering [Imieliński and Lipski 1984], i.e. truth in all clean instances (as opposed to taking the glb), is also co-NP-hard. This result, combined with the reduction provided by Theorem 1, tells us that, among the HCF programs in Datalog, not, the cleaning programs are hard.

**Proposition 5.** Skeptical query answering from cleaning programs is co-NP-complete.

It is possible to obtain a non-disjunctive, stratified cleaning program when matching functions are similarity preserving or MDs are non-interacting. In these cases, the cleaning program has a single stable model, computable in polynomial time, which confirms via cleaning programs a similar result in [Bertossi, Kolahi and Lakshmanan 2011].

### 6 Declarative Swoosh ER: The Union Case

Swoosh, a generic approach to entity resolution [Benjelloun et al. 2009], considers a general match function, $\text{Match}(\cdot, \cdot)$, taking values true or false; and a general merge function, $\mu$ (that would be the matching function of the previous sections). They mostly work at the record level, but the approach can be presented in terms of database tuples [Bertossi, Kolahi and Lakshmanan 2011, section 7].

More precisely, we consider a finite set $\text{Rec}$ of tuples, i.e. ground atoms of the form $R(a)$, for a relational predicate $R(A_1, \ldots, A_n)$, where the $A_i$ are attributes, with domains $\text{Dom}_{A_i}$. For $r_1, r_2 \in \text{Rec}$, $\text{Match}(r_1, r_2)$ takes the value true if the $r_1, r_2$ match; otherwise, false. In the former case, the actual matching is the tuple $\mu(r_1, r_2) \in \text{Rec}$.

When $\text{Match}$ and $\mu$ have the ICAR properties (idempotency, commutativity, associativity and representativity), there is a natural domination partial order on $\text{Rec}$, the merge domination: $r_1 \preceq_S r_2$ if $\text{Match}(r_1, r_2) = \text{true}$ and $\mu(r_1, r_2) = r_2$ [Benjelloun et al. 2009], as we did in Section 2. Similarly, domination can be extended to a partial order $\preceq_S$ on database instances. Given an instance $D$, its entity resolution is defined as the (unique) instance $D'$ that satisfies the conditions: (a) $D' \subseteq D$. (b) $D \preceq_S D'$. (c) No strict subset of $D'$ satisfies the first two conditions [Benjelloun et al. 2009]. Here, $D$ is the merge closure of $D$, i.e. the smallest set of tuples such that includes $D$, and for every $r_1, r_2 \in D$, when $\text{Match}(r_1, r_2) = \text{true}$, also $\mu(r_1, r_2) \in D$.

There is a particular, but still common and broad, class of match and merge functions that is based on union of values. This is the union-case for Swoosh (UC Swoosh), on which we concentrate in the rest of this section. In it, attribute values are represented as sets of finer granularity values, like objects. If $S_1, S_2$ are sets of values for attribute $A$, they are merged via a local merge function $\mu_A$ defined by $\mu_A(S_1, S_2) := S_1 \cup S_2$. The “global” merge function $\mu$ can be defined in terms of the local merge functions $\mu_A$. The match function can also be defined in terms of local, component-based match functions. The resulting merge and match functions satisfy the ICAR properties [Benjelloun et al. 2009; Bertossi, Kolahi and Lakshmanan 2011].

**Example 4.** Consider the instance $D$ below. Attribute $A$ takes as values finite sets of elements from the domain of an underlying, lower-level attribute $A$. E.g. $a_1, a_2 \in \text{Dom}_A \setminus \{a_1, a_2\} \in \text{Dom}_A$. (Similarly for attribute $B$.)

<table>
<thead>
<tr>
<th>$R(D)$</th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a_1}</td>
<td>{b_1}</td>
<td></td>
</tr>
<tr>
<td>{a_2}</td>
<td>{b_2}</td>
<td></td>
</tr>
<tr>
<td>{a_3}</td>
<td>{b_3}</td>
<td></td>
</tr>
</tbody>
</table>

Two tuples match when the values for attribute $A$ match, which happens when there is a pair of values in the $A$-sets that match: For values $S_1, S_2$ for $A$, $\text{Match}(S_1, S_2)$ holds when there are $r_1 \in S_1, r_2 \in S_2$ with $\text{Match}_A(r_1, r_2) = \text{true}$, where $\text{Match}_A$ is a lower-level match function.

Assume $\text{Match}_A(a_1, a_2)$, $\text{Match}_A(a_2, a_3)$ are true. The following is an ER process starting from $D$:

$$R(D') = \{a_1, a_2\} \{b_1, b_2\} \{a_2, a_3\} \{b_2, b_3\} \Rightarrow$$

<table>
<thead>
<tr>
<th>ER(D)</th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a_1, a_2, a_3}</td>
<td>{b_1, b_2, b_3}</td>
<td></td>
</tr>
</tbody>
</table>

Here, we are not using tuple identifiers and we are also getting rid of dominated tuples, as Swoosh does. However, if we had tuple identifiers, keeping them along the ER process, the final instance above would have had three identical tuples, modulo the tuple id.

### 6.1 Special cleaning programs for UC-Swoosh

In this section we use ASPs for the declarative specification of UC Swoosh. In this union case, an attribute, say $A$, can take as a value a whole, finite set of values for an underlying, lower-level attribute $A$. Consequently, in this case the ASPs have to be able to represent sets and sets operations, such as set union. For this purpose we use an extension of disjunctive logic programs with stable model semantics that supports function terms and set terms, with built-in functions for their manipulation [Calimeri et al. 2008; Calimeri et al. 2009].

In this extension of ASP, basic terms are constants and variables, and complex terms, like functional, list and set terms are inductively defined: for terms $t_1, \ldots, t_n$:

1. A functional term is of the form $f(t_1, \ldots, t_n)$, where $f$ is a function symbol.
2. A list term has any of the forms: (a) $[t_1, \ldots, t_n]$, (b) $[b|t]$, where $b$ is a term, and $t$ is a list term.
3. A set term is of the form $\{t_1, \ldots, t_n\}$, where the $t_i$ do not contain any variables. Some functional terms, called built-in functions, have predefined meaning. They are prefixed with #, and we use them below, for sets, lists, membership, and union [Calimeri et al. 2009].

Given a database instance $D$, the swoosh-program $\Pi^{UCS}(D)$ that follows captures UC Swoosh. It contains the rules 1.-4. below:

1. For every atom $R(s) \in D$, $\Pi^{UCS}(D)$ contains a fact of the form $R'(\overline{s})$. For every attribute $A$ of $R$, that takes finite sets of values from an underlying domain $\text{Dom}_A$, facts of the form $\text{Match}_A(a_1, a_2)$, with $a_1, a_2 \in \text{Dom}_A$ (cf. Example 4).
2. Two tuples in $R$ match whenever for attributes $A_i$ of $R$, $1 \leq i \leq n$, there exists a pair of values, one in each of the set values for $A_i$ that match. Hence, for every attribute $A_i$, the rule:

$$R'(\#\text{Union}(S_1, S_2)) \leftarrow R'(S_1), R'(S_2), \#\text{Member}(A_1, S_1^i) \#\text{Member}(A_2, S_2^i), \text{Match}_A(a_1, a_2), S_1^i \neq S_2^i.$$
Here, $S^1 = [S^1_1, \ldots, S^1_n]$, a list of variables; similarly for $S^2$. $R'(\#\text{Union}(S^1, S^2))$ is an abbreviation for the componentwise union, namely: $R'(\#\text{Union}(S^1_1, S^2_1), \ldots, \#\text{Union}(S^1_n, S^2_n))$. The $S^1_1$, $S^2_1$, $A_1$, $A_2$ are variables, whereas in $A_i$, the attribute is fixed. Notice that these rules both specify the match function based on the elements of the set values for attributes, and also the result of the merge.

3. A rule defining tuple domination:

\[ \text{Dominated}(S^1) \leftarrow R'(S^1), R'(S^2), \#\text{Union}(S^1, S^2) = S^2, S^1 \neq S^2. \]

4. A predicate that collects the result of the ER process:

\[ \text{Er}(S) \leftarrow R'(S), \text{not Dominated}(S). \]

The facts in 1. correspond to the elements of the initial instance, and the pairs of low-level attributes values that match. The merge closure of the instance is obtained with $\text{Er}$, we would have a similar, additional rule for it. This program, containing set terms and operations, can be run with DLV-Complex [Calimeri et al. 2009].

Example 5. (example 4 continued) The specific rules are:

1. $R'([a_1], [b_1])$. $R'([a_2], [b_2])$. $R'([a_3], [b_3])$.

\[ \text{Match}_{2}(a_1, a_2). \text{Match}_{3}(a_2, a_3). \]

2. $R'((\#\text{Union}(S^1_1, S^1_2), \#\text{Union}(S^1_3, S^1_4)) \leftarrow R'(S^1_1, S^2_1), R'(S^1_2, S^2_2), \#\text{Member}(A_1, S^1_1), \#\text{Member}(A_2, S^1_2), \text{Match}_{2}(A_1, A_2), (S^1_1, S^2_1) \neq (S^1_2, S^2_2). \]

In this case we are matching via attribute $A$. If we also used $B$, we would have a similar additional rule for it.

3. $\text{Dominated}(S^1_1, S^2_1) \leftarrow R'(S^1_1, S^2_1, R'(S^1_2, S^2_2), (\#\text{Union}(S^1_1, S^1_2), \#\text{Union}(S^2_1, S^2_2)) = (S^2_1, S^2_2), (S^1_1, S^2_1) \neq (S^1_2, S^2_2).$

4. $\text{Er}(S_1, S_2) \leftarrow R'(S_1, S_2), \text{not Dominated}(S_1, S_2)$.

This program, containing set terms and operations, can be run with DLV-Complex [Calimeri et al. 2009].

The answer set programs for UC Swoosh we just introduced are rather ad hoc for this case. However, it is possible to obtain them as special cases of our general ASP approach to ER via MDs developed in Section 3.1 (c.f. [Bahmani et al. 2011, appendix B]). The connection is made possible by the treatment of the UC Swoosh as a special case of MD-based ER developed in [Bertossi, Kalahi and Lakshmanan 2011].

7 Conclusions

In this work we have introduced and developed a declarative approach to ER. It is based on MDs, that can be used to specify details related to ER objectives, like matchings of attribute values when other values are similar. Our work provides a declarative, model-theoretic specification of the process of enforcement of those MDs. The intended clean instances obtained from a dirty instance, become the stable models of a specification given by a cleaning ASP.

We provided a declarative specification for a usually procedural process. It can be executed for ER and clean query answering using a standard ASP solver. Our focus has been on fundamental questions, e.g. required expressive power, complexity issues, and capturing of clean instances. We have made important steps towards those goals, and created the basis for further improvements, e.g. for clean query answering.

Our ASPs can be automatically generated from MDs, and run on ASP solvers. Indeed, we used DLV and DLV-Complex. The programs run and terminate as expected. Our current solution is correct and executable, but there is room for further optimizations.

Optimizations can be of two kinds, “methodological” and “program-related”. In the former category we find, e.g. the development of a more efficient alternative to our approach to clean query answering that computes the glb via rule (2), which involves the free cartesian product of the attribute domains. In this direction, we mentioned “manifold programs” just as a conceptually interesting approach to investigate. In the second category of optimizations we find possible transformations of our ASPs that may lead to more efficient executions. Optimizations of this kind have been developed for repair programs for CQA [Caniupan and Bertossi 2010; Eiter et al. 2008].

We should emphasize that data cleaning and CQA are different problems. For the former, the main goal is to compute a clean instance, as determined by MDs. For the latter, the main goal is obtaining semantically correct query answers. Furthermore, MDs are not (static) ICs. In principle, we could see clean instances as repairs, treating MDs similarly to static FDs. However, none of the existing repair semantics captures the matchings based on MDs with MFs.

We are not aware of any other declarative, ASP-based approach to ER. It should be mentioned that in [Arasu, Ré, and Suciu 2009], Datalog is used for identifying groups of tuples that could be merged. However, they do not do the merging (a main contribution in our approach) or use MDs.

Interesting extensions and applications of our work are discussed in [Bahmani et al. 2011, section 7], e.g. to: (a) virtual data integration systems, (b) combination of ER and database repairs, and (c) Swoosh with negative rules [Whang, Benjelloun and Garcia-Molina 2009].

Acknowledgements: Research funded by NSERC Discovery, NSERC/IBM CRDPJ/371084-2008, and the BIN NSERC Strategic Network on BI (ADC05). We are grateful to Wolfgang Faber, Francesco Calimeri, Paul Fodor, Francesco Ricca, and Alex Brik for valuable information on extensions of ASP. We appreciate the excellent feedback received from anonymous reviewers.
References


