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 similarity conditions 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 $\mathcal{R} = \{R(A, B)\}$, with a predicate R with attributes A and B. The following symbolic expression

$$R[A] \approx R[A] \longrightarrow R[B] \doteq R[B],$$

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_1 and t_2 have similar – actually equal – values for attribute A, but the values for B are different.

$R(D)$	A	B
t_1	a_1	b_1
t_2	a_1	b_2
t_3	a_2	b_3
t_4	a_3	b_4

$R(D')$	A	B
t_1	a_1	b_1
t_2	a_1	b_1
t_3	a_2	b_5
t_4	a_3	b_5

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 $R[B] \approx R[B] \rightarrow R[A] \doteq R[A]$, 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 Kiringa 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 Σ 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 $\mathcal{C}(D, \Sigma)$. *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 $\mathcal{C}(D, \Sigma)$, 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 $\mathcal{G}$, 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 $\mathcal{C}(D, \Sigma)$ of clean instances for D w.r.t. Σ. For each instance D and set Σ 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 $\mathcal{C}(D, \Sigma)$.

The *cleaning program* $\Pi(D, \Sigma)$ axiomatizes the class $\mathcal{C}(D, \Sigma)$. Hence reasoning from/with the program amounts to reasoning with the full class $\mathcal{C}(D, \Sigma)$. 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; Caniupan and Bertossi 2010; Franconi 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 Kiringa 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 $\mathcal{R}$ 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 $R \in \mathcal{R}$ 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_is are different, and different predicates do not share attributes. However, different attributes may share the same domain.

An instance D for $\mathcal{R}$ is a finite set of ground atoms (or tuples) of the form $R(c_1, \ldots, c_n)$, with $R \in \mathcal{R}$, $c_i \in Dom_{A_i}$. 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 $R \in \mathcal{R}$ 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 $\mathcal{A}$ is a list of attributes of predicate R, then $t^D[\mathcal{A}]$ denotes the tuple identified by t, but restricted to the attributes in $\mathcal{A}$. We assume that tuple identifiers are unique across the entire instance.

Schema $\mathcal{R}$ determines a language $L(\mathcal{R})$ of first-order

(FO) predicate logic. A conjunctive query is a formula of $L(\mathcal{R})$ of the form $\mathcal{Q}(\bar{x}): \exists \bar{y}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$, where $P_i \in \mathcal{R}$, $\bar{x} = (\cup_i \bar{x}_i) \smallsetminus \bar{y}$ is the list free variables of the query, say $\bar{x} = x_1 \cdots x_k$. An answer to $\mathcal{Q}$ in instance D is a sequence $\langle a_1, \ldots, a_k \rangle \in U^k$ that makes $\mathcal{Q}$ true in D, denoted $D \models \mathcal{Q}[a_1, \ldots, a_k]$. $\mathcal{Q}(D)$ denotes the set of answers to $\mathcal{Q}$ 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 $\mathcal{R}$ with predicates $R_1[\bar{L}_1], R_2[\bar{L}_2]$, with lists of attributes $\bar{L}_1, \bar{L}_2$, resp., a *matching dependency* (MD) [Fan et al. 2009] is an expression of the form

$$\varphi: \quad R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \longrightarrow R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]. \quad (1)$$

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_1^j, X_2^j, are *pairwise comparable* in the sense that they share the same data domain Dom_j 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 Σ of MDs, a pair of instances (D, D') satisfies Σ 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. Σ). 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, there is a binary *matching function* (MF) $m_A : Dom_A \times Dom_A \to Dom_A$, 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 MFs 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 φ, 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, $t_1^{D_0}[A_1], t_2^{D_0}[A_2]$, to make them both equal to $m_A(t_1^{D_0}[A_1], t_2^{D_0}[A_2])$. 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, Σ be a set of MDs, and $\varphi : R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \to R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]$ be an MD in Σ. 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* φ on t_1, t_2 in instance D, denoted $(D, D')_{[t_1,t_2]} \models \varphi$, if

(a) $t_1^D[\bar{X}_1] \approx t_2^D[\bar{X}_2]$, but $t_1^D[\bar{Y}_1] \neq t_2^D[\bar{Y}_2]$;

(b) $t_1^{D'}[\bar{Y}_1] = t_2^{D'}[\bar{Y}_2] = m_A(t_1^D[\bar{Y}_1], t_2^D[\bar{Y}_2])$; and

(c) D, D' take the same values on every other tuple and attribute. ■

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

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

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: $t_1 \preceq t_2$ iff $t_1[A] \preceq_A t_2[A]$, for each attribute A. This in turn can be lifted to a *relation-level partial order*: $D_1 \sqsubseteq D_2$ 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: $D \sqsubseteq D'$. If instances are all "reduced" by eliminating tuples that are dominated by others, the set of reduced instances with $\sqsubseteq$ forms a lattice. That is, we can compute the *glb* and the *lub* of every finite set of instances w.r.t. $\sqsubseteq$. This is a useful result that allows us to compare sets of query answers w.r.t. $\sqsubseteq$. Indeed, the set of *clean answers* to a query $\mathcal{Q}$ from instance D w.r.t. Σ is formally defined by $Clean_\Sigma^D(\mathcal{Q}) := glb_\sqsubseteq\{\mathcal{Q}(D') \mid D' \in \mathcal{C}(D, \Sigma)\}$ [Bertossi, Kolahi and Lakshmanan 2011].

We will use logic programs Π in $Datalog^{\vee, not}$, 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

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

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

A subset M of HB is a model of Π if it satisfies $gr(\Pi)$, i.e.: For every ground rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m$, $not \; N_1, \ldots, \; not \; N_k$ of $gr(\Pi)$, if $\{P_1, \ldots, P_m\} \subseteq M$ 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 Π if it is a model of Π, and Π has no model that is properly contained in M. $MM(\Pi)$ denotes the class of minimal models of Π. Now, for

$S \subseteq HB(\Pi)$, transform $gr(\Pi)$ into a new, positive program $gr(\Pi)^S$ (i.e. without not), as follows: Delete every rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k$ for which $\{N_1, \ldots, N_k\} \cap S \neq \emptyset$. Next, transform each remaining rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k$ into $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m$. Now, S is a *stable model* of Π if $S \in MM(gr(\Pi)^S)$. Every stable model of Π is also a minimal model of Π.

3 Declarative MD-Based ER

We start by showing that clean query answering is a non-monotonic process.

Example 2. Consider the instance D and the MD ϕ:

$R(D)$	$name$	$phone$	$addr$
t_1	$John\ Doe$	(613)7654321	$Bank\ St.$
t_2	$Alex\ Smith$	(514)1234567	$10\ Oak\ St.$

ϕ: $R[phone, addr] \approx R[phone, addr] \rightarrow R[addr] \doteq R[addr]$.

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

Now, suppose that D is updated into D':

$R(D')$	$name$	$phone$	$addr$
t_1	$John\ Doe$	(613)7654321	$Bank\ St.$
t_2	$Alex\ Smith$	(514)1234567	$10\ Oak\ St.$
t_3	$J.Doe$	7654321	$25\ Bank\ St.$

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

$R(D'')$	$name$	$phone$	$addr$
t_1	$John\ Doe$	(613)7654321	$25\ Bank\ St.$
t_2	$Alex\ Smith$	6131234567	$10\ Oak\ St.$
t_3	$J.Doe$	7654321	$25\ Bank\ St.$

Now, $Clean^{D'}_{\{\phi\}}(\mathcal{Q}) = \mathcal{Q}(D'') = \{\langle 25\ Bank\ St.\rangle\}$. Clearly, $\mathcal{Q}(D) \not\subseteq \mathcal{Q}(D')$, 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-monotonic chase procedure. More precisely, given an instance D_0 and a set Σ of MDs, we propose a logic program $\Pi(D_0, \Sigma)$ whose stable models correspond to the (D_0, Σ)-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, $Prec$, in the cleaning program, and explicitly impose an order on $Prec$ via program constraints.

3.1 Cleaning programs for MDs

Let D_0 be a given, possibly dirty initial instance w.r.t. a set Σ 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 $R'_i(T, \bar{Z})$, where T is a variable for the tuple identifier attribute, and $\bar{Z}$ is a list of variables standing for the (ordinary) attribute values of R_i.

For every attribute A in the schema, with domain Dom_A, the built-in ternary predicate M_A represents the MF m_A, i.e. $M_A(a, a', a'')$ means $m_A(a, a') = a''$. $X \preceq_A Y$ is used as an abbreviation for $M_A(X, Y, Y)$. 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, $=_A$. For lists of variables $\bar{Z}_1 = \langle Z^1_1, \ldots Z^n_1\rangle$ and $\bar{Z}_2 = \langle Z^1_2, \ldots Z^n_2\rangle$, $\bar{Z}_1 \preceq \bar{Z}_2$ denotes the conjunction $Z^1_1 \preceq_{A_1} Z^1_2 \wedge \ldots \wedge Z^n_1 \preceq_{A_n} Z^n_2$. Moreover, for each attribute A, there is a built-in binary predicate $\approx_A$. For two lists of variables $\bar{X}_1 = \langle X^1_1, \ldots X^l_1\rangle$ and $\bar{X}_2 = \langle X^1_2, \ldots X^l_2\rangle$ representing comparable attribute values, $\bar{X}_1 \approx \bar{X}_2$ denotes the conjunction $X^1_1 \approx_1 X^1_2 \wedge \ldots \wedge X^l_1 \approx_l X^l_2$.

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

1. For every tuple (id) $t^{D_0} = R_j(\bar{a})$, the fact $R'_j(t, \bar{a})$.

2. For each MD ϕ_j: $R_1[X_1] \approx R_2[X_2] \rightarrow R_1[A_1] \doteq R_2[A_2]$, the program rules:

$$Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \vee NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \leftarrow R'_1(T_1, \bar{X}_1, Y_1),\ R'_2(T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2.$$

$$OldVersion_{R_i}(T_1, \bar{Z}_1) \leftarrow R'_i(T_1, \bar{Z}_1),\ R'_i(T_1, \bar{Z}'_1),\ \bar{Z}_1 \preceq \bar{Z}'_1,\ \bar{Z}_1 \neq \bar{Z}'_1. \quad (i = 1, 2)$$

$$\leftarrow NotMatch_{\phi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2), not\ OldVersion_{R_1}(T_1, \bar{Z}_1), not\ OldVersion_{R_2}(T_2, \bar{Z}_2).$$

In these rules, the $\bar{X}_i$s are lists of variables corresponding to lists of attributes on the LHS of the MD, whereas the Y_is are single variables corresponding to the attribute on the RHS of the MD. Also, the $\bar{Z}_i$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 $OldVersion_{R_i}$ contains different versions of every tuple (id) in relation R_i 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 $R_i'(t, \bar{a})$ 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 $R_i'(t, \bar{a})$ for a tuple identifier t has been replaced by a newer version $R_i'(t, \bar{a}')$, with $\bar{a} \preceq \bar{a}'$, due to a matching.

The program constraint in 2. above states that if: (a) we have "live", never replaced versions of two tuples (ids) 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 ϕ_j are the same, say R_1, the first rule becomes symmetric w.r.t. every two atoms $R_1'(t_1, \bar{a}_1)$ and $R_1'(t_2, \bar{a}_2)$ that satisfy the body of the rule. We need to make sure that if the matching takes place for these two tuples, then both $Match_{\phi_j}(t_1, \bar{a}_1, t_2, \bar{a}_2)$ and $Match_{\phi_j}(t_2, \bar{a}_2, t_1, \bar{a}_1)$ exist. Thus, for every such MD, we need a rule of the following form

$$Match_{\phi_j}(T_2, \bar{X}_2, Y_2, T_1, \bar{X}_1, Y_1) \ \leftarrow \\ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2).$$

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

$$R_1'(T_1, \bar{X}_1, Y_3) \ \leftarrow \ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \\ M_j(Y_1, Y_2, Y_3).$$

$$R_2'(T_2, \bar{X}_2, Y_3) \ \leftarrow \ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \\ M_j(Y_1, Y_2, Y_3).$$

4. For every two matchings applicable to different versions of a tuple with a given identifier, we record in $Prec$ 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.

$$Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1', T_3, \bar{Z}_3) \ \leftarrow \\ Match_{\phi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2), Match_{\phi_k}(T_1, \bar{Z}_1', T_3, \bar{Z}_3), \\ \bar{Z}_1 \preceq \bar{Z}_1', \ \bar{Z}_1 \neq \bar{Z}_1'.$$

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 $Match$ 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 $Prec$ 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 $Prec$.

$$Prec(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2, T_1, \bar{X}_1, Y_1, T_3, \bar{X}_3, Y_3) \ \leftarrow \\ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \\ Match_{\phi_k}(T_1, \bar{X}_1, Y_1, T_3, \bar{X}_3, Y_3), M_k(Y_1, Y_3, Y_4), \ Y_1 \neq Y_4.$$

This rule says that, in case (a ground version of) a tuple $\langle T_1, \bar{X}_1, Y_1 \rangle$ participates in two matching, via MDs ϕ_j and ϕ_k, and the tuple changes according to ϕ_k, as captured by the last two body atoms that use ϕ_k's matching function M_k, then the matching via ϕ_k must come after the matching via ϕ_j. By this same rule, the reverse $Prec$-order could also be true, but we will disallow having both by imposing conditions on $Prec$, 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 $Prec$-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 $Prec$ 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 $Prec$, unless they both do not change the tuple.

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

$$Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1, T_2, \bar{Z}_2) \ \leftarrow \\ Match_{\phi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2).$$

$$\leftarrow Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1', T_3, \bar{Z}_3), \\ Prec(T_1, \bar{Z}_1', T_3, \bar{Z}_3, T_1, \bar{Z}_1, T_2, \bar{Z}_2), \\ (T_1, \bar{Z}_1, T_2, \bar{Z}_2) \neq (T_1, \bar{Z}_1', T_3, \bar{Z}_3).$$

$$\leftarrow Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1', T_3, \bar{Z}_3), \\ Prec(T_1, \bar{Z}_1', T_3, \bar{Z}_3, T_1, \bar{Z}_1'', T_4, \bar{Z}_4), \\ not \ Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1'', T_4, \bar{Z}_4).$$

Notice that we do not use ***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 R_i^c the latest version of each tuple for every predicate R_i; they are used to form the clean instances.

$$R_i^c(T_1, \bar{Z}_1) \ \leftarrow \ R_i'(T_1, \bar{Z}_1), \ not \ OldVersion_{R_i}(T_1, \bar{Z}_1).$$

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 $Match_\varphi$ with the MDs φ, 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 had. An alternative to our approach would be to build a *single* program that can be used with any schema and finite

set of MDs associated to it. Such a program is bound to be much more complex than those, specific but still generic, that we are proposing here.

Example 3. Consider relation $R(A, B)$, and set Σ with:

$$\phi_1 : R[A] \approx R[A] \rightarrow R[B] \doteq R[B],$$
$$\phi_2 : R[B] \approx R[B] \rightarrow R[B] \doteq R[B].$$

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

$M_B(b_1, b_2, b_{12})$,
$M_B(b_2, b_3, b_{23})$,
$M_B(b_1, b_{23}, b_{123})$.

$R(D_0)$	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

Enforcing Σ 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.

D_0	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

$\Rightarrow_{\phi_1}$

D_1	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{12}
t_3	a_3	b_3

D_0	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

$\Rightarrow_{\phi_2}$

D'_1	A	B
t_1	a_1	b_1
t_2	a_2	b_{23}
t_3	a_3	b_{23}

$\Rightarrow_{\phi_1}$

D'_2	A	B
t_1	a_1	b_{123}
t_2	a_2	b_{123}
t_3	a_3	b_{23}

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

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

2. $Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee NotMatch_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx X_2,\ Y_1 \neq Y_2.$

$Match_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee NotMatch_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ Y_1 \approx Y_2,\ Y_1 \neq Y_2.$

$Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow Match_{\phi_1}(T_2, X_2, Y_2, T_1, X_1, Y_1).$ (similarly for $Match_{\phi_2}$)

$OldVersion_R(T_1, \bar{Z}_1) \leftarrow R'(T_1, \bar{Z}_1),\ R'(T_1, \bar{Z}'_1), \bar{Z}_1 \preceq \bar{Z}'_1,\ \bar{Z}_1 \neq \bar{Z}'_1.$

$\leftarrow NotMatch_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2), not\ OldVersion_R(T_1, X_1, Y_1), not\ OldVersion_R(T_2, X_2, Y_2).$ (similarly for $NotMatch_{\phi_2}$)

3. $R'(T_1, X_1, Y_3) \leftarrow Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_B(Y_1, Y_2, Y_3).$

$R'(T_1, X_1, Y_3) \leftarrow Match_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_B(Y_1, Y_2, Y_3).$

4. $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y'_1, T_3, X_3, Y_3) \leftarrow Match_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_k}(T_1, X_1, Y'_1, T_3, X_3, Y_3), Y_1 \preceq Y'_1,\ Y_1 \neq Y'_1.$ (with $1 \leq j, k \leq 2$)

5. $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \leftarrow Match_{\phi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_k}(T_1, X_1, Y_1, T_3, X_3, Y_3),\ M_B(Y_1, Y_3, Y_4), Y_1 \neq Y_4.$ (with $1 \leq j, k \leq 2$)

7. $R^c(T_1, X_1, Y_1) \leftarrow R'(T_1, X_1, Y_1), not\ OldVersion_R(T_1, X_1, Y_1).$

Program $\Pi(D_0, \Sigma)$ has two stable models, whose R^c-atoms are shown below:

$M_1 = \{..., R^c(t_1, a_1, b_{12}), R^c(t_2, a_2, b_{12}), R^c(t_3, a_3, b_3)\},$
$M_2 = \{..., R^c(t_1, a_1, b_{123}), R^c(t_2, a_2, b_{123}), R^c(t_3, a_3, b_{23})\}.$

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 $\mathcal{C}(D_0, \Sigma)$ 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. Σ are exactly the restrictions of the elements of $SM(\Pi(D_0, \Sigma))$ to schema $\mathcal{R}^c$. ■

The restriction of the stable models to the relational schema $\mathcal{R}^c$ 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, Σ)-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'_i, $OldVersion_{R_i}$, 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 $Match_{\phi_j}$, $NotMatch_{\phi_j}$, and $Prec$. 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, Σ)-clean instance D_S. To construct D_S, we let $t^{D_S} = \bar{a}$ for every relation R_i and every tuple identifier t of relation R_i such that $R^c_i(t, \bar{a}) \in S$. Then using the $Match_{\phi_j}$ atoms in S and also the relative order of these matchings in $Prec$, 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 $Prec$. 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 D_S. ■

4 Query Answering

We can use the cleaning program $\Pi(D_0, \Sigma)$ to compute the clean answers to a query $\mathcal{Q}$ posed to D_0, as defined in Section 2. The clean answers were defined by taking into account the underlying lattices, as the *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 $\mathcal{Q}(x_1, \ldots, x_n)$, 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 $\Pi(\mathcal{Q})$ can be obtained from φ, using a standard transformation [Lloyd 1987]. It contains an answer predicate $Ans_{\mathcal{Q}}(\bar{x})$ to collect the answers to $\mathcal{Q}$, and rules defining it, of the form $Ans_{\mathcal{Q}}(\bar{x}) \leftarrow B(\bar{x}')$, where the Bs are conjunctions of literals (i.e. atoms or negations $not\ A$ thereof). The R-atoms in $\mathcal{Q}$, with $R \in \mathcal{R}$, are replaced in $\Pi(\mathcal{Q})$ by R^c-atoms.

We can obtain reduced answer sets (cf. Section 2) by adding two new rules to $\Pi(\mathcal{Q})$:[1]

$$\begin{aligned} Ans^r_{\mathcal{Q}}(\bar{x}) &\leftarrow Ans_{\mathcal{Q}}(\bar{x}),\ not\ Dominated_{\mathcal{Q}}(\bar{x}). \\ Dominated_{\mathcal{Q}}(\bar{x}) &\leftarrow Ans_{\mathcal{Q}}(\bar{y}),\ \bar{x} \preceq \bar{y},\ \bar{x} \neq \bar{y}. \end{aligned}$$

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

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

$$Ans^{exp}_{\mathcal{Q}}(S_i) := \{\bar{b} \mid \bar{b} \preceq \bar{a}, \text{ for some } \bar{a} \in Ans^r_{\mathcal{Q}}(S_i)\}.$$

$Ans^{exp}_{\mathcal{Q}}(S_i)$ contains all the atoms in $Ans^r_{\mathcal{Q}}(S_i)$ and everything below w.r.t. the $\preceq$ lattice. Since $Ans^r_{\mathcal{Q}}(S_i)$ is finite, $Ans^{exp}_{\mathcal{Q}}(S_i)$ is also finite, because we consider finite lattices.

Proposition 1. Let D_0 be an instance, Σ be a set of MDs, and $\mathcal{Q}$ be a query. Let $SM(\Pi(D_0, \Sigma, \mathcal{Q})) = \{S_1, \ldots, S_m\}$, then $glb_{\sqsubseteq}\{Ans^r_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\} = Red_{\sqsubseteq}(\bigcap\{Ans^{exp}_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\})$. ■

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

$$Ans^{exp}_{\mathcal{Q}}(\bar{y}) \leftarrow Ans^r_{\mathcal{Q}}(\bar{x}),\ \bar{y} \preceq \bar{x},\ Dom_{\mathcal{L}}(\bar{y}). \qquad (2)$$

Here, $Dom_{\mathcal{L}}(\cdot)$ is stands for the cartesian product of the finite domains Dom_A for the local lattices L_A.

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

We have just described a way to compute, by means of the *downward expanded* programs, the clean answers to a query $\mathcal{Q}$. 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 *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 Π, an MF program for Π, say $MF(\Pi)$, extends Π by collecting brave or skeptical atomic consequences from what would have been Π -now a part of $MF(\Pi)$- and using them for further processing by $MF(\Pi)$.

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

$$\begin{aligned} glb_{\preceq}(\bar{x}, U) &\leftarrow U = \#Union(\{\bar{y}\}, U'), glb_{\preceq}(\bar{u}, U'), \\ &\quad\ \bar{x} = glb^t_{\preceq}(\bar{u}, \bar{y}). \\ glb_{\preceq}(\bar{x}, \{\bar{x}\}) &\leftarrow Dom(\bar{x}). \\ PAns_{\mathcal{Q}}(\bar{x}) &\leftarrow glb_{\preceq}(\bar{x}, \{\bar{x}_1, \ldots, \bar{x}_m\}), Ans_{\mathcal{Q}}(\bar{x}_1)^{S_1}, \\ &\quad\ \ldots, Ans_{\mathcal{Q}}(\bar{x}_m)^{S_m}. \\ Dominated^p_{\mathcal{Q}}(\bar{x}) &\leftarrow PAns_{\mathcal{Q}}(\bar{y}),\ \bar{x} \preceq \bar{y},\ \bar{x} \neq \bar{y}. \\ CAns(\bar{x}) &\leftarrow PAns_{\mathcal{Q}}(\bar{x}),\ not\ Dominated^p_{\mathcal{Q}}(\bar{x}). \end{aligned}$$

Here, $glb_{\preceq}(\bar{x}, U)$ is a binary predicate that says that tuple $\bar{x}$ is the $glb_{\preceq}$ of set U; and is defined by recursion and associativity: $glb_{\preceq}(\{\bar{y}\} \cup U') = glb^t_{\preceq}(\bar{y}, glb_{\preceq}(U'))$. $glb^t_{\preceq}(\bar{u}, \bar{y})$ is a function that produces the $glb_{\preceq}$ 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 *glb* of a set. The domain predicate, *Dom*, is associated to the cartesian product of the finite attribute domains involved.

The third rule computes the *pre-answers* by combination into the *glb* $\bar{x}$ of braves answers obtained from the $Ans_{\mathcal{Q}}(\bar{x}_i)^{S_i}$. 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

[1] Notice that $\preceq$ in the second rule is defined in terms of the built-in relations M_A.

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 $Datalog^{\vee,not,s}$, we denote the subclass of programs in $Datalog^{\vee,not}$ 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.

$Datalog^{\vee,not}$ extends the classes $Datalog$, $Datalog^{not,s}$, and $Datalog^{not}$ 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]. $Datalog^{\vee,not,s}$ 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 $Datalog^{not}$ (without stratified negation) may have multiple stable models.

The (likely) higher expressive power of $Datalog^{\vee,not}$ 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 $\Pi \in Datalog^{\vee,not}$, i.e. if A is true in all the stable models of Π, is Π_2^P-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 $Datalog^{\vee,not,s}$. ■

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 $Datalog^{\vee,not,s}$ is the same as for programs with unstratified negation, i.e. for the class $Datalog^{\vee,not}$, i.e. Π_2^P-complete [Eiter and Gottlob 1995; Dantsin et al. 2001; Gelfond and Leone 2002].

Repair programs for CQA under ICs, also belong to the class $Datalog^{\vee,not,s}$ [Caniupan 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 Π_2^P-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 that is non-disjunctive. To undertake this task, we need some terminology.

Let $\Pi \in Datalog^{\vee,not}$, and Π^g be its ground version. The *dependency graph*, $DG(\Pi^g)$, is a directed graph whose nodes are literals of Π^g. There is an arc from L_1 to L_2 iff there is a rule in Π^g where L_1 appears positive in the body and L_2 appears in the head. Π is *head-cycle free* (HCF) iff $DG(\Pi^g)$ has no cycle through two literals that belong to the head of a same rule [Ben-Eliyahu and Dechter 1994].

HCF programs in $Datalog^{\vee,not}$ 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 $Datalog^{not}$. We have:

Proposition 3. Every cleaning program $\Pi(D_0, \Sigma)$ is HCF, and hence can be transformed into an equivalent non-disjunctive program in $Datalog^{not}$. ■

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 Σ), amounts to checking if D' is contained in stable model of $\Pi(D_0, \Sigma)$, since the stable models also contain atoms other than $\mathcal{R}$-atoms. Equivalently, 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 D' is a clean instance (just checking stability, i.e. if $(D', D') \models \Sigma$ 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 $Datalog^{not}$ 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 Σ of MDs, and a FO query $\mathcal{Q}(\bar{x})$, deciding if a tuple $\bar{c}$ is a clean answer to $\mathcal{Q}$ from an instance D_0 belongs to the class *co-NP* (in the size of D_0).[2] ■

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.

[2] To be precise, we have to use program $\Pi(D_0, \Sigma, \mathcal{Q})$ expanded with rule (2), which actually adds to D_0 the extension of $Dom_{\mathcal{L}}$. 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* [Imielinski 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 $Datalog^{\vee,not}$, 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*, $Match(\cdot,\cdot)$, taking values *true* or *false*; and a general *merge function*, μ (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 Rec of tuples, i.e. ground atoms of the form $R(\bar{a})$, for a relational predicate $R(A_1,\ldots,A_n)$, where the A_i are attributes, with domains Dom_{A_i}. For $r_1, r_2 \in Rec$, $Match(r_1, r_2)$ takes the value *true* if the r_1, r_2 match; otherwise, *false*. In the former case, the actual matching is the tuple $\mu(r_1, r_2) \in Rec$.

When $Match$ and μ have the *ICAR* properties (idempotency, commutativity, associativity and representativity), there is a natural domination partial order on Rec, the *merge domination*: $r_1 \leq_s r_2$ iff $Match(r_1, r_2) = true$ 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) $D' \subseteq \bar{D}$. (b) $\bar{D} \preceq_s D'$. (c) No strict subset of D' satisfies the first two conditions [Benjelloun et al. 2009]. Here, $\bar{D}$ is the *merge closure* of D, i.e. the smallest set of tuples such that includes D, and for every $r_1, r_2 \in \bar{D}$, when $M(r_1, r_2) = true$, also $\mu(r_1, r_2) \in \bar{D}$.

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 μ_A defined by $\mu_A(S_1, S_2) := S_1 \cup S_2$. The "global" merge function μ can be defined in terms of the local merge functions μ_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 $\underline{A}$. E.g. $a_1, a_2 \in Dom_{\underline{A}}, \{a_1, a_2\} \in Dom_A$. (Similarly for attribute B.) 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, $Match_A(S_1, S_2)$ holds when there are $v_1 \in S_1, v_2 \in S_2$ with $Match_{\underline{A}}(v_1, v_2) = true$, where $Match_{\underline{A}}$ is a lower-level match function.

$R(D)$	A	B
	$\{a_1\}$	$\{b_1\}$
	$\{a_2\}$	$\{b_2\}$
	$\{a_3\}$	$\{b_3\}$

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

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

$\Rightarrow$

$ER(D)$	A	B
	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$

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, $\underline{A}$. 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 $t_1, ..., t_n$: 1. A *functional term* is of the form $f(t_1, ..., t_n)$, where f is a function symbol. 2. A *list term* has any of the forms: (a) $[t_1, ..., t_n]$; (b) $[h|t]$, where h is a term, and t is a list term. 3. A *set term* is of the form $\{t_1, ..., t_n\}$, 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 $R(\bar{s}) \in D$, $\Pi^{UCS}(D)$ contains a fact of the form $R'(\bar{s})$. For every attribute A of R, that takes finite sets of values from an underlying domain $Dom_{\underline{A}}$, facts of the form $Match_{\underline{A}}(a_1, a_2)$, with $a_1, a_2 \in Dom_{\underline{A}}$ (cf. Example 4).

2. Two tuples in $\mathcal{R}$ 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:

$$R'(\#Union(\bar{S}^1, \bar{S}^2)) \leftarrow R'(\bar{S}^1),\ R'(\bar{S}^2), \#Member(A_1, S_i^1)\ \#Member(A_2, S_i^2), Match_{\underline{A_i}}(A_1, A_2),\ \bar{S}^1 \neq \bar{S}^2.$$

Here, $\bar{S}^1 = [S_1^1, \ldots, S_n^1]$, a list of variables; similarly for $\bar{S}^2$. $R'(\#Union(\bar{S}^1, \bar{S}^2))$ is an abbreviation for the componentwise union, namely: $R'(\#Union(S_1^1, S_1^2), \ldots, \#Union(S_n^1, S_n^2))$. The S_j^1, S_j^2, A_1, A_2 are variables, whereas in $\underline{A_i}$, 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:

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

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

$$Er(\bar{S}) \leftarrow R'(\bar{S}),\ not\ Dominated(\bar{S}).$$

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 rules in 2. By the properties of match and merge functions for the UC, dominated tuples in the merge closure of D can be eliminated via merge domination, which is specified by rule 3. Rule 4. collects those tuples of the merge closure D that are not dominated.

It is easy to verify that the program $\Pi^{UCS}(D)$ is stratified. Then, it has a single stable model that can be computed bottom-up in polynomial time in the size of D. This model, restricted to predicate Er, coincides with the ER procedurally computed in [Benjelloun et al. 2009], where it was shown that the ICAR properties make the ER computation tractable. In consequence, our declarative approach to UC Swoosh is in line with the results in [Benjelloun et al. 2009].

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

1. $R'(\{a_1\}, \{b_1\})$. $R'(\{a_2\}, \{b_2\})$. $R'(\{a_3\}, \{b_3\})$. $Match_{\underline{A}}(a_1, a_2)$. $Match_{\underline{A}}(a_2, a_3)$.

2. $R'(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) \leftarrow R'(S_1^1, S_2^1), R'(S_1^2, S_2^2), \#Member(A_1, S_1^1), \#Member(A_2, S_1^2), Match_{\underline{A}}(A_1, A_2),\ (S_1^1, S_2^1) \neq (S_1^2, S_2^2)$.

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

3. $Dominated(S_1^1, S_2^1) \leftarrow R'(S_1^1, S_2^1),\ R'(S_1^2, S_2^2),\ (\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) = (S_1^2, S_2^2),\ (S_1^1, S_2^1) \neq (S_1^2, S_2^2)$.

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

This program, containing set terms and operations, can be run with *DLV-Complex* [Calimeri et al. 2009].[3] ■

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 C]). The connection is made possible by the treatment of the UC Swoosh as a special case of MD-based ER developed in [Bertossi, Kolahi and Lakshmanan 2011].

[3] http://www.mat.unical.it/dlv-complex. Cf. [Bahmani et al. 2011, appendix B] for the DLV code.

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

Abiteboul, S., Hull, R. and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

Arasu, A., Ré, Ch. and Suciu, D. Large-Scale Deduplication with Constraints Using Dedupalog. Proc. ICDE 2009, pp. 952-963.

Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. Proc. PODS 1999, pp. 68-79.

Arenas, M., Bertossi, L. and Chomicki, J. Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, 2003, 3(4-5):393-424.

Bahmani, Z., Bertossi, L., Kolahi, S. and Lakshmanan, L. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs (Extended Version). 2001. http://www.scs.carleton.ca/~bertossi/papers/extZeinab11.pdf

Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In *Semantics in Databases*, Springer LNCS 2582, 2003, pp. 7-33.

Ben-Eliyahu, R. and Dechter, R. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics in Artificial Intelligence*, 1994, 12:53-87.

Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Euijong Whang, S. and Widom, J. Swoosh: A Generic Approach to Entity Resolution. *VLDB Journal*, 2009, 18(1):255-276.

Bertossi, L. Consistent Query Answering in Databases. *ACM Sigmod Record*, 2006, 35(2):68-76.

Bertossi, L. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management, Morgan & Claypool, 2011.

Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. Proc. ICDT 2011, 12 pages.

Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. To appear in *Theory of Computing Systems* journal, 2012.

Bleiholder, J. and Naumann, F. Data Fusion. *ACM Computing Surveys*, 2008, 41(1).

Brewka,G., Eiter, T. and Truszczynski, M. Answer Set Programming at a Glance. *Comm. of the ACM*, 2011, 54(12), pp. 93-103.

Calimeri, F. Cozza, S. Ianni, G. and Leone, N. Computable Functions in ASP: Theory and Implementation. Proc. ICLP 2008, Springer LNCS 5366, pp. 407-424.

Calimeri, F. Cozza, S. Ianni, G. and Leone, N. An ASP System with Functions, Lists, and Sets. Proc. LPNMR 2009, Springer LNCS 5753, pp. 483-489.

Caniupan, M. and Bertossi, L. The Consistency Extractor System: Answer Set Programs for Consistent Query Answering in Databases. *Data & Knowledge Engineering*, 2010, 69(6):545-572.

Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer, 1989.

Chomicki, J. Consistent Query Answering: Five Easy Pieces. Proc. ICDT 2007, Springer LNCS 4353, pp. 1-17.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 2001, 33(3):374-425.

Eiter, T., Gottlob, G. and Mannila, H. Disjunctive Datalog. *ACM Trans. Database Syst.*, 1997, 22(3):364-418.

Eiter, T. and Gottlob, G. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Math. and Artif. Intell.*, 1995, 15(3-4):289-323.

Eiter, T., Fink, M., Greco, G. and Lembo, D. Repair Localization for Query Answering from Inconsistent Databases. *ACM Trans. Database Syst.*, 2008, 33(2).

Elmagarmid, A., Ipeirotis, P. and Verykios, V. Duplicate Record Detection: A Survey. *IEEE Transactions in Knowledge and Data Engineering*, 2007, 19(1):1-16.

Faber, W. and Woltran, S. Manifold Answer-Set Programs and Their Applications. In *Gelfond Festschrift*, Springer LNAI 6565, 2011, pp. 44-63.

Fan, W. Dependencies Revisited for Improving Data Quality. Proc. PODS 2008, pp. 159-170.

Fan, W., Jia, X., Li, J. and Ma, S. Reasoning about Record Matching Rules. *PVLDB*, 2009, 2(1):407-418.

Franconi, E., Laureti Palma, A., Leone, N., Perri, S. and Scarcello, F. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. Proc. LPAR 2001, Springer LNCS 2250, pp. 561-578.

Gardezi, J., Bertossi, L. and Kiringa, I. Matching Dependencies with Arbitrary Attribute Values: Semantics, Query Answering and Integrity Constraints. Proc. EDBT/ICDT International Workshop on Logic in Databases (LID 2011).

Gelfond, G. and Lifschitz, V. Logic Programs with Classical Negation. Proc. ICLP 1990, pp. 579-597.

Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9(3/4):365-386.

Gelfond, M. and Leone, N. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 2002, 138(1-2):3-38.

Greco, G., Greco, S. and Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Trans. Knowledge and Data Eng.*, 2003, 15(6):1389-1408.

Imielinski, T. and Lipski, W. Incomplete Information in Relational Databases. *Journal ACM*, 1984, 31(4):761-791.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 2006, 7(3):499-562.

Lloyd, J. *Foundations of Logic Programming*. Springer, 1987, 2nd. edition.

Whang, S.E., Benjelloun, O. and Garcia-Molina, H. Generic Entity Resolution with Negative Rules. *VLDB Journal*, 2009, 18(6):1261-1277.