Declarative Entity Resolution via Matching Dependencies and Answer Set Programs (extended version)

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 2997]. 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], and more formally investigated in [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 symbolic expression

$$R[A] \approx R[A] \longrightarrow \ R[B] \doteq R[B]), \qquad (1)$$

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, $a_2 \approx a_3$, and $a_2 \not\approx a_1 \not\approx a_3$.

On the LHS below we have the extension $R(D)$ 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

That is, the MD above does not hold in D, so it is a *dirty* instance. After applying the MD, we could get the instance D' on the RHS, where the values for B have been identified. Notice that 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 $R[B] \approx R[B] \longrightarrow \ R[A] \doteq R[A])$, 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, e.g. the similarity $b_1 = b_1$ in D'. ■

A *dynamic semantics* of MDs was introduced in [Fan et al. 2009], in the sense that it requires a pair of database instances: a first one where the similarities hold and second one where the matchings are enforced, like D and D' in Example 1. This example also shows that the MDs, as introduced in [Fan et al. 2009], do not specify how the matchings have to be made. We could even pick up a new value, like b_5 above, for the common value. This semantics was refined and extended in [Bertossi, Kolahi and Lakshmanan 2011], using a *matching function* (MF) to guide the matchings, for each of the participating attribute domains. The MFs induce a lattice-theoretic structure on these domains [Bertossi, Kolahi and Lakshmanan 2011]. An alternative dynamic semantics was introduced in [Gardezi, Bertossi and Kiringa 2011]. It does not appeal to matching functions, but match-

ings have to be mandatory (as also in [Bertossi, Kolahi and Lakshmanan 2011]) and *minimal*, i.e., a minimum number of changes to attribut values need to be applied in order to satisfy the MDs.

In this work we revisit the approach to ER via MDs introduced in [Bertossi, Kolahi and Lakshmanan 2011], that uses matching functions. 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 matching functions. 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). The clean answers 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 in [Bertossi, Kolahi and Lakshmanan 2011]. 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 new declarative programming paradigm [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. Given a concrete, properly encoded instance, I, for the problem, the models of the combined specification $\mathcal{G} \cup I$ become the solutions for instance I. The solutions can be computed using a general underlying *solver*. In principle, different kinds of logical formalisms can be used [Marek, Niemela, Truszczynski 2011]. However, most commonly, and also historically, the logical formalism is the one provided by *(extended) disjunctive logic programs with stable model semantics* [Gelfond and Lifschitz 1991], also commonly called *answer set programs* [Gelfond and Leone 2002].

In this work, we introduce answer set programs, in the form of *disjunctive Datalog* [Eiter, Gottlob and Mannila 1997], to specify the class $\mathcal{C}(D, \Sigma)$ of clean instances for D w.r.t. Σ. Actually, for each instance D and set Σ of MDs, we show how to build an answer set program $\Pi(D, \Sigma)$ whose stable models (in the sense of the stable model semantics) 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, the clean query answers can be obtained from the original instance D by skeptical or cautious reasoning from the program.

Answer set programs have been successfully used before in the area of *consistent query answering* (CQA) in databases [Arenas, Bertossi, and Chomicki 1999; Bertossi 2006; Chomicki 2007; Bertossi 2011], in the form of *repair programs*, that specify the *repairs* of an 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]. They can be used for CQA [Caniupan and Bertossi 2010], i.e., to compute certain answers w.r.t. the class of all repairs. However, *MDs cannot be treated as classical ICs.* For the particular framework used in this work, the matching functions and the lattice-theoretic structure of the domains, with the induced domination order, create a scenario that is substantially different from the one encountered in database repairs w.r.t. classical ICs. New challenges arise that cannot be solved by reusing repair programs or what we know about them. Furthermore, the semantics of MDs is also quite different from the one 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 their syntactic structures, properties associated with them, and complexity results. In particular, we show that their expressive power is appropriate for our application in ER, and they are in line with the computational complexity of computing clean instances and clean query answers. This also allows us to establish complexity properties for checking clean instances. We also show how to use cleaning programs with the skeptical (or cautious) semantics for the computation of clean answers from the original database, with a similar data complexity.

The Swoosh approach to ER was proposed in [Benjelloun et al. 2009], as a generic specification of ER procedures. In particular, the "union-case" of entity resolution is investigated. In that framework, individual records (or attribute values in them) can be seen as sets of triples of the form $(id, attr, value)$, i.e. as *objects*. An ER step basically matches values by producing their union. In this way, the resulting value dominates, in terms of information contents, the two original values. This is a common and important scenario for ER. In [Bertossi, Kolahi and Lakshmanan 2011], the Swoosh's union-case was reconstructed in the MD framework. In [Benjelloun et al. 2009] the Swoosh approach to ER is presented in procedural terms. 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 point to possible extensions of our work and to ongoing and future research. To conclude, in Section 8, we obtain a few final conclusions. Proofs of results, and also examples with the *DLV* system [Leone et al. 2006] that we have used for

small experiments with cleaning programs, can be found in the appendices.

2 Preliminaries

2.1 Relational databases

We consider relational schemas $\mathcal{S}$ consisting of a possibly infinite data domain U, a finite set $\mathcal{R}$ of database predicates, say R, each of a fixed, finite arity, and a set $\mathcal{B}$ of built-in predicates, like $=, \neq$, etc. Each $R \in \mathcal{R}$, of some arity n, has n attributes, say $A_1, \ldots, A_n$, each of them with a domain $Dom_{A_i} \subseteq U$. For simplicity, and without loss of generality, we may assume that all the A_i are different, and different predicates do not share attributes. However, different attributes may share the same domain.

An instance D for schema $\mathcal{S}$ can be seen as a finite set of ground atoms of the form $R(c_1, \ldots, c_n)$, with $c_i \in Dom_{A_i}$. $R(c_1, \ldots, c_n)$ is also called a *tuple*. In this work, we will assume that tuples have identifiers, as in Example 1. Tuple identifiers allow us to compare the extensions of the same predicate in different instances, which is necessary since we are updating instances by changing attribute values (and never inserting or deleting tuples).

Tuple identifiers can be accommodated by adding an extra attribute, say T, to each predicate $R \in \mathcal{R}$, so that the tuples take the form $R(t, c_1, \ldots, c_n)$, with t a value for T. We assume that for each predicate, T acts as a key, i.e., all other attributes functionally depend upon T. Most of the time we leave the tuple identifier, or we use the tuple identifier to denote the whole tuple, e.g., t for denoting the whole tuple $R(t, c_1, \ldots, c_n)$. More precisely, if t is a tuple identifier in an instance D, then t^D denotes the entire database 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 $\mathcal{A}$.

Schema $\mathcal{S}$ determines a language $L(\mathcal{S})$ of first-order (FO) predicate logic. A conjunctive query is a formula of $L(\mathcal{S})$ of the form $\mathcal{Q}(\bar{x}) \colon \exists \bar{y}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$, where $P_i \in \mathcal{R} \cup \mathcal{B}$, $\bar{x} = (\cup_i \bar{x}_i) \smallsetminus \bar{y}$ is the list free variables of the query. If $\bar{x} = x_1 \cdots x_k$, an answer to the query in instance D is a sequence $\langle a_1, \ldots, a_k \rangle \in U^k$, such that $D \models \mathcal{Q}[a_1, \ldots, a_k]$, i.e., that makes the query true in D when the variables in it are replaced by $a_1, \ldots, a_k$. We denote with $\mathcal{Q}(D)$ the set of answers to $\mathcal{Q}$ in D.

Notice that a query can be seen as a mapping that sends instances to instances [Abiteboul, Hull, and Vianu 1995], of a possibly different schema. E.g., the resulting schema may have the single, new predicate $\mathcal{Q}$, of arity k (and with k attributes from $\mathcal{S}$), and $\mathcal{Q}(D)$ would be an instance of the new schema.

2.2 Matching dependencies for ER

Given a relational schema $\mathcal{S}$ 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 a formula of the form (omitting quantifiers)

$$\varphi \colon \ R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \rightarrow R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]. \qquad (2)$$

Here, where $\bar{X}_1, \bar{Y}_1$ are sublists of $\bar{L}_1$, and $\bar{X}_2, \bar{Y}_2$ sublists of $\bar{L}_2$. The lists $\bar{X}_1, \bar{X}_2$ 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. Similarly, the corresponding attributes in Y_1, Y_2 also share a domain. Actually, (2) can be seen as an abbreviation for the formula

$$\varphi \colon \ \bigwedge_j R_1[X_1^j] \approx_j R_2[X_2^j] \rightarrow \bigwedge_k R_1[Y_1^k] \doteq R_2[Y_2^k].$$

This MD intuitively states that if for an R_1-tuple t_1 and an R_2-tuple t_2 in 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 we need to make the values $t_1^D[\bar{Y}_1]$ and $t_2^D[\bar{Y}_2]$ identical. This will result in another instance D', where $t_1^{D'}[\bar{Y}_1], t_2^{D'}[\bar{Y}_2]$ have been updated in a way that $t_1^{D'}[\bar{Y}_1] = t_2^{D'}[\bar{Y}_2]$ holds. Without loss of generality, we assume that the list of attributes on the right-hand side of MDs contain only one attribute (see [Fan et al. 2009]).

If we have a set Σ of MDs, like the one in (2), 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 side, an instance D is *stable* if $(D, D) \models \Sigma$ [Fan et al. 2009].

In order to *enforce* an MD on two tuples and make the 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) $m_A : Dom_A \times Dom_A \rightarrow Dom_A$, such that the value $m_A(a, a')$ is used to replace two values $a, a' \in Dom_A$ whenever the two values need to be made equal. Following [Bertossi, Kolahi and Lakshmanan 2011; Benjelloun et al. 2009], we expect MFs to be idempotent, commutative, and associative.

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 the elements of Dom_A is defined as follows: For every $a, a' \in Dom_A$, $a \preceq_A a'$ whenever $m_A(a, a') = a'$. The *lub* operator with respect to this partial order coincides with m_A: $lub_{\preceq_A}\{a, a'\} = m_A(a, a')$. We also assume the existence of the greatest lower bounds, $glb(a, a')$ [Bertossi, Kolahi and Lakshmanan 2011].

In the rest of this work, especially in answer set programs, we will assume by default that similarity relations and matching functions are built-in (evaluable, embedded) relations with fixed extensions and the desired properties. Only occasionally, we may introduce rules for specifying them and enforcing their properties.

A chase-based semantics for data cleaning (or entity resolution) with matching dependencies 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., $t_1^{D_0}[\bar{X}_1] \approx t_2^{D_0}[\bar{X}_2]$,[1] and apply a matching function 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

[1] but not its RHS

$m_A(t_1^{D_0}[A_1], t_2^{D_0}[A_2])$. 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 $\varphi : R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \rightarrow R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]$ 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'. We say that 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' agree on every other tuple and attribute value. ■

Definition 2. For an instance D_0 and a set of MDs Σ, 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. However, for the special cases of (a) similarity-preserving MFs (i.e., $a \approx a'$ implies $a \approx m_A(a', a''), \forall a, a', a''$); and (b) interaction-free MDs (i.e., each attribute may appear in either right or left-hand side of MDs in Σ), there is a unique clean instance for a dirty instance D. In general, $\mathcal{C}(D, \Sigma)$ denotes the set of clean instances for instance D w.r.t. Σ.

2.3 Semantic domination and clean answers

The relation $a \preceq_A a'$ can be thought of in terms of relative information contents [Bertossi, Kolahi and Lakshmanan 2011]. This notion has been investigated in *domain theory* [Gunter and Scott 1990], in the context of *semantic-domination lattices*. Domain-level partial order $\preceq_A$ 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 $\forall t_1 \in D_1\ \exists t_2 \in D_2\ t_1 \preceq t_2$.

It can be observed that when a tuple t^D in instance D is updated to $t^{D'}$ in instance D' by enforcing an MD and applying a matching function, we have $t^D \preceq t^{D'}$. Moreover, 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 greatest lower bound (*glb*) and the least upper bound (*lub*) of every finite set of instances w.r.t. partial order $\sqsubseteq$. 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]. The clean answers are similar to the *certain answers* [Imielinski and Lipski 1984], but here the partial order is brought into the definition. Deciding clean answers is *co-NP*-complete [Bertossi, Kolahi and Lakshmanan 2011].

2.4 Disjunctive Datalog with SMS

We will use logic programs Π in $Datalog^{\vee,not}$, i.e., (function-free) disjunctive Datalog programs with weak negation [Gelfond and Lifschitz 1991; Eiter, Gottlob and Mannila 1997]. They contain a finite number of rules of the form

$$A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k, \quad (3)$$

where $0 \leq n, m, k$, and A_i, P_j, N_s are (positive) atoms. The terms in these atoms are constants or variables. 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 $gr(\Pi)$, that is: 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, take $S \subseteq HB(\Pi)$, and 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 $S \in MM(gr(\Pi)^S)$. It is well known that a stable model of Π is a minimal model of Π.

The expressive power of $Datalog^{\vee,not}$ (cf. Section 5) has been useful and necessary for applications to database repairs and CQA [Caniupan and Bertossi 2010] due to the inherently rather high complexity of CQA [Bertossi 2011].

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, $Clean_{\{\phi\}}^D(\mathcal{Q}) = \mathcal{Q}(D) = \{\langle Bank\ St.\rangle\}$.

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 a database instance D_0 and a set of MDs Σ, we develop a logic program $\Pi(D_0, \Sigma)$, whose stable models correspond to (D_0, Σ)-clean instances that can be obtained by enforcing matching dependencies in a chase-like procedure.

The 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 that will give us clean instances.

Program $\Pi(D_0, \Sigma)$ specifies clean instances by explicitly eliminating, using program denial constraints, instances that are the result of illegal applications of matching dependencies. 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 matching applications 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 record pairs of matchings in an auxiliary relation, $Prec$, in the cleaning program of Section 3.1, and explicitly impose an order on the auxiliary relation using program constraints.

3.1 Cleaning programs for MDs

Let D_0 be a given initial instance that is possibly dirty w.r.t. a given 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 an additional attribute used for the tuple identifier, 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, we use a built-in ternary predicate M_A that represents the matching function m_A. Here, $M_A(a, a', a'')$ means $m_A(a, a') = a''$, $a, a', a'' \in Dom_A$. We write $X \preceq_A Y$ 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 two lists of variables $\bar{Z}_1 = \langle Z_1^1, \ldots Z_1^n\rangle$ and $\bar{Z}_2 = \langle Z_2^1, \ldots Z_2^n\rangle$, we write $\bar{Z}_1 \preceq \bar{Z}_2$ to denote the conjunction $Z_1^1 \preceq_{A_1} Z_2^1 \wedge \ldots \wedge Z_1^n \preceq_{A_n} Z_2^n$.

Moreover, for each attribute A, there is a built-in binary predicate $\approx_A$. For two lists of variables $\bar{X}_1 = \langle X_1^1, \ldots X_1^l\rangle$ and $\bar{X}_2 = \langle X_2^1, \ldots X_2^l\rangle$ representing comparable attribute values, we write $\bar{X}_1 \approx \bar{X}_2$ to denote the conjunction $X_1^1 \approx_1 X_2^1 \wedge \ldots \wedge X_1^l \approx_l X_2^l$.

These predicates are built-in in the sense that they have a fixed extension, and their atoms can be evaluated at request by some built-in mechanism. Of course, in some applications we may not want to treat M_A or $\approx_A$ as built-ins. In those cases they could be further specified by means of additional program rules, in particular to enforce their expected properties (cf. Section 2.2).

The program $\Pi(D_0, \Sigma)$ contains the rules in **1.-9.** below:[2]

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

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

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

$$\leftarrow NotMatch_{\phi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2),$$
$$not\ OldVersion_1(T_1, \bar{Z}_1), not\ OldVersion_2(T_2, \bar{Z}_2).$$

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

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 right-hand side of the MD. Also, the $\bar{Z}_i$s are lists of variables corresponding to all attributes in a tuple.

These rules are used to enforce the MD whenever the necessary similarities hold for two tuples. The first rule specifies that in that case, a matching may or may not take place, but the latter is acceptable only if one of the involving tuples is used for another matching and replaced by a newer version. This is enforced using the second rule, which is a program constraint (it has the effect of filtering out stable models where the conjunction in its body becomes true).

Notice that 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 third rule specifies when an atom $R'_i(t, \bar{a})$ for a tuple identifier t has been replaced by a newer version $R'_i(t, \bar{a}')$, where $\bar{a} \preceq \bar{a}'$, due to a matching application. For convenience, below we refer to the various atoms associated with a given tuple identifier t as versions of the tuple identifier t.

[2] All the capitalized arguments of predicates are variables (e.g. X) or lists thereof (e.g. $\bar{X}$). We use this notation to make the association with attributes or lists thereof in MDs easier.

Notice that, when the two predicates appearing in ϕ_j are the same, say R_1, then 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 therefore 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 matching dependency, 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 attribute on 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 need to record the relative order of the two matchings. This is done through an auxiliary predicate, $Prec$. The matching that is applied to the smaller version of the tuple according to partial order $\preceq$ has to precede the other matching.

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

We need similar rules (four rules in total) for the cases where the common tuple identifier variable T_1 appears in different components of the two $Match$ predicates (cf. rules 4. in Example 3 below).

5. Each version of a tuple identifier can participate in more than one matching only if at least one of them does not change the tuple. For every two matchings applicable to a single version of a tuple identifier, we need to record the relative order of the two matchings in $Prec$. The matching that produces a new version for the tuple has to come after the other matching. If both of the two matchings do not produce a new version of the tuple, they can be applied in any order, and there is no need to explicitly record their relative order in $Prec$.

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

As in 4. above, we need four rules of this form, for different possible appearances of the common variable T_1. This rule will never allow two matchings that will produce incomparable versions of a tuple w.r.t. $\preceq$, because we ensure that $Prec$ is antisymmetric. (Cf. rules 6.-8. below that specify the properties of $Prec$.) 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 relation:

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

7. Program constraints for making $Prec$ antisymmetric:

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

8. Program constraints for making $Prec$ transitive:

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

9. Finally, rules to collect in R_i^c the latest version of each tuple for every predicate R_i.

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

These rules are used to form the clean instances.

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 labels (the subindices) to indicate the latter.

Example 3. Consider relation $R(A, B)$, and set Σ consisting of the following MDs:

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

Assume that in the dirty instance D_0 below, exactly the following similarities hold: $a_1 \approx a_2$, $b_2 \approx b_3$. Moreover, the matching functions act 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 will result in the following 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: (although built-in, we make the extension of MFs explicit)

1. $a_1 \approx a_2.$ $\quad b_2 \approx b_3.$
$R'(t_1, a_1, b_1).$ $\quad R'(t_2, a_2, b_2).$ $\quad R'(t_3, a_3, b_3).$
$M_B(b_1, b_2, b_{12}).$ $M_B(b_2, b_3, b_{23}).$ $M_B(b_1, b_{23}, b_{123}).$

2. $Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee$
$NotMatch_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow$
$R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx X_2,\ Y_1 \neq Y_2.$
$Match_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee$
$NotMatch_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow$
$R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ Y_1 \approx Y_2,\ Y_1 \neq Y_2.$
$Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow$
$Match_{\phi_1}(T_2, X_2, Y_2, T_1, X_1, Y_1).$
(simmilarly for $Match_{\phi_2}$)
$\leftarrow NotMatch_{\phi_1}(T_2, X_2, Y_2, T_1, X_1, Y_1),$
$not\ OldVersion(T_1, X_1, Y_1),$
$not\ OldVersion(T_2, X_2, Y_2).$
(simmilarly for $NotMatch_{\phi_2}$)
$OldVersion(T_1, \bar{Z}_1) \leftarrow R'(T_1, \bar{Z}_1),\ R'(T_1, \bar{Z}'_1),$
$\bar{Z}_1 \preceq \bar{Z}'_1,\ \bar{Z}_1 \neq \bar{Z}'_1.$

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 $1 \leq j, k \leq 2$)

As mentioned above, we may need four similar rules, when the common tuple identifer T_1 appears in different components of the two match predicates; e.g one of them is:

$Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_3, X_3, Y_3, T_1, X_1, Y'_1) \leftarrow$
$Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2),$
$Match_{\phi_2}(T_3, X_3, Y_3, T_1, X_1, Y'_1),\ Y_1 \preceq Y'_1,\ Y_1 \neq Y'_1.$

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 $1 \leq j, k \leq 2$)

As for the rules in 4., we have four similar rules, for cases with different appearances of common variable T_1; e.g.,

$Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \leftarrow$
$Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2),$
$Match_{\phi_2}(T_3, X_3, Y_3, T_1, X_1, Y_1),$
$M_B(Y_1, Y_3, Y_4),\ Y_1 \neq Y_4.$

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

7. $\leftarrow Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X'_1, Y'_1, T_3, X_3, Y_3),$
$Prec(T_1, X'_1, Y'_1, T_3, X_3, Y_3, T_1, X_1, Y_1, T_2, X_2, Y_2),$
$(T_1, X_1, Y_1, T_2, X_2, Y_2) \neq (T_1, X'_1, Y'_1, T_3, X_3, Y_3).$

8. $\leftarrow Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X'_1, Y'_1, T_3, X_3, Y_3),$
$Prec(T_1, X'_1, Y'_1, T_3, X_3, Y_3, T_1, X''_1, Y''_1, T_4, X_4, Y_4),$
$not\ Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X''_1, Y''_1, T_4, X_4, Y_4).$

9. $R^c(T_1, X_1, Y_1) \leftarrow R'(T_1, X_1, Y_1),$
$not\ OldVersion(T_1, X_1, Y_1).$

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 Appendix B. ■

Next we establish that, for an instance D_0 and a set Σ of MDs, the set $SM(\Pi(D_0, \Sigma))$ of the stable models of $\Pi(D_0, \Sigma)$, is in one-to-one corresponds with the set of (D_0, Σ)-clean instances.

Proposition 1. For every (D_0, Σ)-clean instance D_k, we can construct a set of atoms S_{D_k} that is a stable model for the logic program $\Pi(D_0, \Sigma)$. ■

Proposition 2. For every stable model S of the program $\Pi(D_0, \Sigma)$, we can construct a (D_0, Σ)-clean instance D_S. ■

Theorem 1. There is a one-to-one correspondence between $\mathcal{C}(D, \Sigma)$ and the set $SM(\Pi(D, \Sigma)$ of stable models of the cleaning program $\Pi(D, \Sigma)$. More precisely, the clean instances for D w.r.t. Σ are exactly the restrictions of the elements of $SM(\Pi(D, \Sigma))$ to schema $\mathcal{S}$. ■

The restriction of the stable models to the relational schema $\mathcal{S}$ is due to the fact that they have extensions for the auxiliary predicates used in the programs.

4 Query Answering

We would like to 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 2.3. The set of clean answers is defined there, by taking into account the underlying lattices, as the greatest lower bound 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 the skeptical query answering from the cleaning programs.

Given a FO query $\mathcal{Q}(x_1, \ldots, x_n)$, with free variables standing for attributes $A_1, \ldots, A_n$ of $\mathcal{S}$, 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 $R \in \mathcal{S}$, are replaced in $\Pi(\mathcal{Q})$ by R^c-atoms.

We can obtain reduced answer sets (cf. Section 2.3) by adding two new rules to $\Pi(\mathcal{Q})$:

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

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

Next we show how the program $\Pi(D, \Sigma, \mathcal{Q})$ can be modified, so that the clean answers to query $\mathcal{Q}$ can be obtained by running the program under the skeptical semantics.

Given $Ans^r_{\mathcal{Q}}(S_i)$, the set of answers to $\mathcal{Q}$ from the clean instance corresponding to the stable model S_i, we define its *downward expansion* by:

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

That is, $Ans^{exp}_{\mathcal{Q}}(S_i)$ contains all the atoms in $Ans^r_{\mathcal{Q}}(S_i)$ and everything below them w.r.t. the $\preceq$ lattice. Notice that, since $Ans^r_{\mathcal{Q}}(S_i)$ is finite, then $Ans^{exp}_{\mathcal{Q}}(S_i)$ is also finite, because we assume the lattices are finite.

Proposition 3. Let D be an instance, Σ be a set of MDs, and $\mathcal{Q}$ be a query. Let $SM(\Pi(D, \Sigma, \mathcal{Q})) = \{S_1, \ldots, S_m\}$, 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 process) instead of taking the *glb* over all sets $Ans^r_{\mathcal{Q}}(S_i)$. This can be achieved directly through $\Pi(D, \Sigma, \mathcal{Q})$ by adding to it the following rule:

$$Ans^{exp}_{\mathcal{Q}}(\bar{y}) \leftarrow Ans^r_{\mathcal{Q}}(\bar{x}),\ \bar{y} \preceq \bar{x},\ Dom_{\mathcal{L}}(\bar{y}). \qquad (4)$$

Here, $Dom_{\mathcal{L}}(\cdot)$ is a predicate standing for the cartesian product of the finite domains for the local lattices L_A, where A is an attribute of the query schema.

In order to obtain the clean answers to query $\mathcal{Q}$, the query atom $Ans^{exp}_{\mathcal{Q}}(\bar{y})$ can be skeptically answered from this extension of $\Pi(D, \Sigma, \mathcal{Q})$, which has the effect of computing the ground query atoms in the intersection of its stable models.

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 Section 7.1 we discuss an alternative approach to clean query answering.

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 $Datalog^{\vee,not,s}$, we denote the subclass of programs in $Datalog^{\vee,not}$ that have *stratified negation*. For these programs, the set of predicates $\mathcal{P}$ can be partitioned into a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_k$ in such a way that, for every $P \in \mathcal{P}$:

1. If $P \in \mathcal{P}_i$ and predicate Q appears in a head of a rule with P, then $Q \in \mathcal{P}_i$.
2. If $P \in \mathcal{P}_i$ and Q appears positively in the body of a rule that has P in the head, then $Q \in \mathcal{P}_j$, with $j \leq i$.
3. If $P \in \mathcal{P}_i$ and Q appears negatively in the body of a rule that has P in the head, then $Q \in \mathcal{P}_j$, with $j < i$.

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.

$Datalog^{\vee,not}$ (cf. Section 2.4) 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., the set of *facts* or rules of the form (3) with $n = 1, m = k = 0$, and A_1 ground. 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^P_2-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 4. 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, if they provide an expressive power that exceeds the one needed for clean query answering. Secondly, as to 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 where each literal L (i.e., of the form A or $not\ A$, with A an atom) is a node, and 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 its $DG(\Pi^g)$ does not contain directed cycles that go through two literals that belong to the head of a same rule [Ben-Eliyahu and Dechter 1994; Dantsin et al. 2001].

Example 4. Consider the ground program

$$\Pi = \{a \vee b \leftarrow c,\ d \leftarrow b,\ a \vee b \leftarrow e, not\ f\}.$$

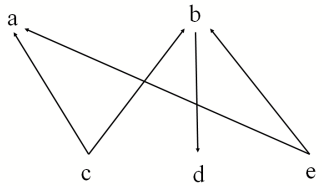

Figure 1: $DG(\Pi)$

Its dependency graph is shown in Figure 1. Π is HCF, because there is no cycle involving both a and b, the atoms that appear in the disjunctive head. ■

Programs in $Datalog^{\vee,not}$ that are HCF 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 5. Every cleaning program $\Pi(D, \Sigma)$ 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 our case, the disjunctive rule

$$\begin{aligned} &Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \vee \\ &\quad NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow \\ &\quad R_1'(T_1, \bar{X}_1, Y_1),\ R_2'(T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2. \end{aligned}$$

gives rise to two rules:

$$\begin{aligned} &Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow R_1'(T_1, \bar{X}_1, Y_1), \\ &\quad R_2'(T_2, \bar{X}_2, Y_2), not\ NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \\ &\qquad\qquad \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2. \end{aligned}$$

and

$$\begin{aligned} &NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow R_1'(T_1, \bar{X}_1, Y_1), \\ &\quad R_2'(T_2, \bar{X}_2, Y_2), not\ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \\ &\qquad\qquad \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2. \end{aligned}$$

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 and Σ), amounts to checking if D' is contained in stable model of $\Pi(D, \Sigma)$, since the stable models also contain atoms other than $\mathcal{S}$-atoms ($\mathcal{S}$ is the relational schema). 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 5 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 $\Pi(D, \Sigma)$ belongs to the class *co-NP*. From this result and Theorem 1, we obtain

Proposition 6. 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 belongs to the class *co-NP*(in the size of D).[3] ■

This result should be contrasted with the *co-NP*-complete data complexity of deciding clean query answers presented

[3] To be precise, we have to use program $\Pi(D, \Sigma, \mathcal{Q}))$ expanded with rule (4), which actually adds to D the extension of $Dom_{\mathcal{L}}$. However, the latter could be left as a fixed parameter.

in [Bertossi, Kolahi and Lakshmanan 2011, Theorem 3]. We have reobtained the membership to *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.

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*, 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 2, tells us that, among the HCF programs in $Datalog^{\vee,not}$, the cleaning programs are hard.

Proposition 7. 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

In [Benjelloun et al. 2009] a generic approach to entity resolution, *Swoosh*, was proposed and developed. A general *match function*, M, and a general *merge function*, μ are considered. 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 $\mathcal{R}$ 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 $r_1, r_2 \in \mathcal{R}$, $M(r_1, r_2)$ takes the value *true* if the r_1, r_2 match; otherwise, *false*. In the former case, the actual matching is the tuple $\mu(r_1, r_2) \in \mathcal{R}$.

When M and μ have the *ICAR* properties (idempotency, commutativity, associativity and representativity), there is a natural domination or subsumption partial order on $\mathcal{R}$, the merge domination: $r_1 \preceq_s r_2$ iff $M(r_1, r_2) = \mathit{true}$ and $\mu(r_1, r_2) = r_2$ [Benjelloun et al. 2009], as we did in Section 2. As outlined there, 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 $M(r_1, r_2) = \mathit{true}$, 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 will concentrate in the rest of this section.

More precisely, 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 5. Consider the instance D below.

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

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, $M_A(S_1, S_2)$ holds when there are $v_1 \in S_1, v_2 \in S_2$ with $m(v_1, v_2) = \mathit{true}$, where m is a lower-level match function.

Assume that $m(a_1, a_2)$ and $m(a_2, a_3)$ hold (are true). The following is an ER process starting from D:

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

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

In this example 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 provide answer set programs (ASPs) for the *declarative specification* of the union-case (UC) of Swoosh. Since we are dealing with attributes that take as values entire set of values, the ASPs have to be able to capture sets and sets operations, such as 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, basic terms are constants and variables, and complex terms including 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*, are predefined, with a fixed meaning. They are prefixed by # in the language. Similarly for predicates, atoms prefixed by # use built-in predicates, with predefined meanings [Calimeri et al. 2009].

Example 6. We can build facts like $Parent(person, \{child_1, \ldots, child_n\})$, to show associations between parents

and children. We can obtain the names of all descendants of a person with the following rules:

$$Ancestor(A, Cs) \leftarrow Parent(A, Cs).$$
$$Ancestor(A, \#Union(Ds, Ss)) \leftarrow Ancestor(A, Ds), \#Member(S, Ds),\ Parent(S, Ss).$$

Here, $\#Union(Ds, Ss)$ and $\#Member$ are built-in predicates with the intended meaning of union and membership (to a set or a list). ■

Given a database instance D, the *swoosh-program* $\Pi^{UCS}(D)$ that follows captures the generic UC-Swoosh approach to ER [Benjelloun et al. 2009]. It contains the following 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 $a_1, a_2 \in Dom_{\underline{A}}$.

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 $\bar{D}$ 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 7. (example 5 continued) The specific rules are:

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

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].[4] ■

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. 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, section 7].

7 Extensions

In this section we point to and present some ideas for interesting extensions to our work that we are currently pursuing or intend to develop.

7.1 Clean QA and manifold programs

In Section 4, on clean query answering, we described a way to compute the clean answers to a query $\mathcal{Q}$ without a separate and off-line gathering of query answers from each of the stable models for later combination via the *glb*. This was achieved via the *downward expanded* programs. The *manifold programs* (MF) [Faber and Woltran 2011] offer another alternative for using a single ASP for the whole task, at least conceptually, since implementation issues seem still to be open.

Given a program Π, an MF program for Π, say $MF(\Pi)$, extends or subsumes Π by collecting atomic brave or skeptical 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 $\Pi(D, \Sigma, \mathcal{Q})$ of the form $Ans_{\mathcal{Q}}(\bar{a})^S$, with $S \in SM(\Pi(D, \Sigma, \mathcal{Q}))$, can be further used by $MF(\Pi(D, \Sigma, \mathcal{Q}))$ to compute the mentioned *glb*s. For this latter task, $MF(\Pi(D, \Sigma, \mathcal{Q}))$ could include rules of the form (we give

[4] http://www.mat.unical.it/dlv-complex Cf. Appendix B. for the DLV code.

a high-level description of them):

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

Here, $glb_{\preceq}(\bar{x}, U)$ is a binary predicate that says that tuple $\bar{x}$ is the $glb_{\preceq}$ of set U. It is defined by recursion and associativity: $glb_{\preceq}(\{\bar{y}\} \cup U') = glb^{t}_{\preceq}(\bar{y}, glb_{\preceq}(U'))$. Here, $glb^{t}_{\preceq}(\bar{u}, \bar{y})$ is a function that produces the $glb_{\preceq}$ (as operation) of two tuples. The first two rules use the extension of ASP with sets and operations with them [Calimeri et al. 2009] that we saw in Section 6.1. They recursively compute the *glb* of a set. The third one 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 last one computes the clean answers by discarding the pre-answers that are dominated by other pre-answers.

7.2 Swoosh with negative rules

In [Whang, Benjelloun and Garcia-Molina 2009], the original Swoosh approach to ER is extended with *negative rules*, that impose constraints on the merge results.

Example 8. Consider the instance D, subject to an ER process under Swoosh's union case:

$R(D)$	$name$	$phone$	$gender$
t_1	$\{Mishael\}$	$\{7654321\}$	$\{\}$
t_2	$\{Michael\}$	$\{\}$	$\{M\}$
t_3	$\{Mishael\}$	$\{7654321\}$	$\{F\}$

Without going into details, by using the original Swoosh, we could obtain the following final result:

$R(D_2)$	$name$	$phone$	$gender$
t_{123}	$\{Mishael,\ Michael\}$	$\{7654321\}$	$\{F, M\}$

However, a *negative rule* prohibiting for a person to be both M and F, would avoid reaching this instance. This might require, as done in [Whang, Benjelloun and Garcia-Molina 2009], to appeal at some points to an external expert, to make the right merge decisions. ■

As shown in Appendix D, it is possible to extend, our ASP-based account of Swoosh by taking into consideration negative rules, and also the use of external experts. The latter is achieved via *HEX* programs, which are extensions of ASP with calls to external sources [Eiter et al. 2005].

7.3 MDs and database repairs

The combination and interaction of database repairs, as found in CQA [Bertossi 2011], and matching dependencies has been initially investigated in [Fan et al. 2011].

The cleaning programs we have presented in this work could be combined in different ways with *repair programs*. The latter are disjunctive programs with stable model semantics, and are used to specify -as stable models- the repairs of a database that fails to satisfy a given set of integrity constraints [Arenas, Bertossi, and Chomicki 2003; Barcelo, Bertossi and Bravo 2003; Greco, Greco and Zumpano 2003; Caniupan and Bertossi 2010].

Example 9. Consider a relational schema with a predicate $P(X, Y)$, and the functional dependency (FD) $X \rightarrow Y$, stating that the first attribute functionally determines the second one. The instance $D = \{P(a, b), P(a, c), P(d, e)\}$ is an inconsistent w.r.t. FD. D has two repairs, i.e., instances that satisfy FD, and make minimal, under set inclusion, the sets of deleted tuples that are required to reestablish consistency. They are $D_1 = \{P(a, b), P(d, e)\}$ and $D_2 = \{P(a, c), P(d, e)\}$.

The repairs of D w.r.t. FD can be specified as the stable models of a logic program that contains a main rule of the form

$$P(x, y, \mathbf{f}) \vee P(x, z, \mathbf{f}) \leftarrow P(x, y), P(x, z), y \neq z.$$

It specifies that whenever the FD is violated, which is captured by the body of the rule, then (only) one of the two tuples involved in the violation has to be deleted (made false), which is captured by the disjunctive head. ■

Other repair policies, e.g., changes of attribute values as opposed to tuple deletions, can also be expressed via repair programs. Repair and cleaning programs could interact in different ways.

7.4 ER and virtual data integration

Doing entity resolution on a virtual data integration system is a challenging problem. A user may not have access to the data sources, and the matchings can be applied only on-the-fly, at query answering time. Something similar happens with violations of global ICs and database repairs.

Actually, this idea was developed in [Bravo and Bertossi 2003], as follows. First, leaving the ICs aside, the *legal*, intended global instances of a virtual data integration system [Lenzerini 2002] can be specified as the stable models of an answer set program. On top of that program, a repair program, fully combined with the former into a single program, computes the repairs of the global instances. In this way, the consistent answers from the integration system can be computed.

A similar approach could be attempted with ER via global MDs. The cleaning program can be combined with the program that specifies the intended instances of the integration system.

8 Conclusions

In this work we have introduced and developed a declarative approach to entity resolution (ER). It is based on matching dependencies (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 given, dirty instance, become the stable models of a specification that takes the form of an answer set program (ASP), a so-called *cleaning program*.

We have provided some first ideas and techniques on how to do clean query answering using the cleaning program. This is a subject that requires more investigation. In general, implementation issues are quite open. We have used the *DLV* system and its extension with sets to run our examples. As with repair programs for consistent query answering, there is room for many optimizations [Caniupan and Bertossi 2010; Eiter et al. 2008] that still have to be investigated.

We have pointed to important extensions and research directions, most importantly to applications in virtual data integration systems, where the system can be specified declaratively using ASP-based specifications. Indeed, explicitly computing clean instances is not practical, rather computing clean query answers on the fly from the ASP is the only realistic option.

Acknowledgements: This research was funded by an NSERC Discovery grant, the NSERC/IBM CRDPJ/371084-2008, and the BIN NSERC Strategic Network on Business Intelligence (project ADC05). We are grateful to Francesco Calimeri, Paul Fodor, Francesco Ricca, and Alex Brik for valuable information on answers set programs with sets and aggregation. Very nice and informative email exchanges with Wolfgang Faber on manifold programs are also greatly appreciated.

References

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

Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. Proc. ACM PODS'99, 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.

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, ACM Press, 2011.

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

Bravo, L. and Bertossi, L. Logic Programs for Consistently Querying Data Integration Systems. Proc. IJCAI 2003, pp. 10-15.

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, 2009, 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 Mathematics and Artificial Intelligence*, 1995, 15(3-4):289-323.

Eiter, T., Ianni, G., Schindlauer, R. and Tompits, V. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. Proc. IJCAI, 2005, pp. 90-96.

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. ACM PODS'08, 2008, pp. 159-170.

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

Fan, W., Li, J., Ma, S., Tang, N. and Yu, W. Interaction Between Record Matching and Data Repairing. Proc. SIGMOD, 2011, pp. 469-480.

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

Gelfond, G. and Lifschitz, V. Logic Programs with Classical Negation. Proc. Int. Conf. Logic Programming, 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 Transactions on Knowledge and Data Engineering*, 2003, 15(6):1389-1408.

Gunter, C.A. and Scott, D.S. Semantic Domains. Chapter 12 in *Handbook of Theoretical Computer Science*, Vol. B, Elsevier, 1990.

Imielinski, T. and Lipski, W. Jr. Incomplete Information in Relational Databases. *J. 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.

Lenzerini, M. Data Integration: A Theoretical Perspective. Proc. ACM PODS 2002, pp. 233-246.

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

Marek, V., Niemela, I. and Truszczynski, M. Origins of Answer Set Programming - Some Background and Two Personal Accounts. In *Nonmonotonic Reasoning. Essays Celebrating its 30th Anniversary*, G. Brewka, V. Marek and M. Truszczynski (eds.). College Publications, 2011.

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

A Proofs

Those auxiliary technical results that are stated in this appendix, but not in the main body of the paper, are numbered in the form $\mathbf{A.n}$, e.g., Lemma A.1.

Proof of Proposition 1: Let D_k be a (D_0, Σ)-clean instance. That is, there are instances $D_1, \ldots, D_{k-1}$ such that, for every $j \in [1,k]$, $(D_{j-1}, D_j)_{[t_1,t_2]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers t_1, t_2. We construct S_{D_k}, a set of atoms over relations of the logic program $\Pi(D_0, \Sigma)$, as follows.

- For every instance D_j, $j \in [0,k]$ and every tuple identifier t of relation R_i, S_{D_k} contains an atom $R_i'(t, \bar{a})$, where $t^{D_j} = \bar{a}$.
- For every tuple identifier t of relation R_i, S_{D_k} contains an atom $R_i^c(t, \bar{a})$, where $t^{D_k} = \bar{a}$ for clean instance D_k.
- For every instance D_j, $j \in [0, k-1]$ and every tuple identifier t of relation R_i such that $t^{D_j} \neq t^{D_k}$, S_{D_k} contains an atom $OldVersion_i(t, \bar{a})$, where $t^{D_j} = \bar{a}$.
- For every $j \in [0,k]$, tuple identifiers t_1, t_2 and MD φ, such that $(D_{j-1}, D_j)_{[t_1,t_2]} \models \varphi$, S_{D_k} contains an atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$, where $t_1^{D_{j-1}} = \bar{a}_1$ and $t_2^{D_{j-1}} = \bar{a}_2$. If the two relation names appearing in φ are the same, S_{D_k} also contains $Match_\varphi(t_2, \bar{a}_2, t_1, \bar{a}_1)$.
- For every $j, l \in [0,k]$, tuple identifiers t_1, t_2 and MD φ, such that $t_1^{D_j} = \bar{a}_1$, $t_2^{D_l} = \bar{a}_2$, $t_1^{D_j}, t_2^{D_l}$ satisfy the left-hand side similarity of φ but do not satisfy the right-hand side equality, and $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2) \notin S_{D_k}$, S_{D_k} contains $NotMatch_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$.
- For every $j, l \in [0,k]$, tuple identifiers t_1, t_2, t_3 and MDs φ_1, φ_2, such that $(D_{j-1}, D_j)_{[t_1,t_2]} \models \varphi_1$, $(D_{l-1}, D_l)_{[t_1,t_3]} \models \varphi_2$, and $j \leq l$, S_{D_k} contains an atom $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_1, \bar{a}_1', t_3, \bar{a}_3)$, where $t_1^{D_{j-1}} = \bar{a}_1$, $t_2^{D_{j-1}} = \bar{a}_2$ $t_1^{D_{l-1}} = \bar{a}_1'$, and $t_3^{D_{l-1}} = \bar{a}_3$.

It is easy to observe that S_{D_k} is a stable model for the program $\Pi(D_0, \Sigma)$. ■

Proof of Proposition 2: Let S be a stable model for the logic program $\Pi(D_0, \Sigma)$. For every relation R_i and every tuple identifier t of relation R_i such that $R_i^c(t, \bar{a}) \in S$, we let $t^{D_S} = \bar{a}$. To show that D_S is a (D_0, Σ)-clean instance, we need to construct instances $D_1, \ldots, D_k = D_S$, such that, for every $j \in [1,k]$, $(D_{j-1}, D_j)_{[t_1,t_2]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers t_1, t_2. We use the following lemma.

Lemma A.1. The relation $prec$ is a partial order on the set of atoms $\mathcal{M} = \{Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2) \mid Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2) \in S\}$. ■

The lemma easily follows from rules 6, 7, and 9.

Let $\leq$ be any linear order on $\mathcal{M}$ that preserves $prec$. That is, for every two match atoms $Match_{\varphi_j}(t_1, \bar{a}_1, t_2, \bar{a}_2)$ and $Match_{\varphi_l}(t_3, \bar{a}_3, t_4, \bar{a}_4)$, we have $Match_{\varphi_j}(t_1, \bar{a}_1, t_2, \bar{a}_2) \leq Match_{\varphi_l}(t_3, \bar{a}_3, t_4, \bar{a}_4)$ whenever $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_3, \bar{a}_3, t_4, \bar{a}_4)$ holds, and equality holds only when the two atoms are identical.

For every $i \in [1,k]$, $k = |\mathcal{M}|$, we construct instance D_i as follows. Let $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ be the ith smallest atom in the linear order $\leq$. For every tuple identifier t, we let $t^{D_i} = t^{D_{i-1}}$ if $t \neq t_1, t_2$, and we let $t_1^{D_i}, t_2^{D_i}$ be the result of enforcing φ on $\bar{a}_1, \bar{a}_2$. Due to minimality of S, $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ exists only if the left-hand side similarities of the MD φ hold for $\bar{a}_1, \bar{a}_2$, and the right-hand side equality does not hold. We thus have $(D_{i-1}, D_i)_{[t_1,t_2]} \models \varphi$. It remains to show that D_k is a stable instance, and it is actually equal to D_S.

Lemma A.2. Let $S_{R'}$ contain all the R' atoms from the stable model S, i.e., $S_{R'} = \{R_j'(t, \bar{a}) \mid R_j'(t, \bar{a}) \in S\}$. Let $D_{R'}$ be a set of R' atoms constructed from instances $D_0, \ldots, D_k$ above, defined as $D_{R'} = \{R_j'(t, \bar{a}) \mid$

t is an R_j-tuple, and $t^{D_i} = \bar{a}$ for some $i \in [1, k]\}$. ■

Then it holds $S_{R'} = D_{R'}$.

Using Lemma A.2 we can easily show that if D_k is not a stable instance, then S cannot be a stable model of the program. Moreover, since D_k and D_S collect the largest version of each tuple identifier, w.r.t. $\preceq$, from the identical sets of atoms $S_{R'}$ and $D_{R'}$, the two instances should be equal. ■

Proof of Proposition 5: Let us suppose that $\Pi(D, \Sigma)$ is not HCF. Then the program $\Pi(D, \Sigma)$ has a directed cycle in its dependency graph that goes through $Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ and $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ (the atoms that appear in the only disjunctive head), but $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ cannot be involved in a directed cycle since there is no rule in $\Pi(D, \Sigma)$ in which $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ appears in the body of a rule having heads, a contradiction. ■

Definition A.1. A model M of a normal program Π is stable iff M is a minimal model of Π^M, where Π^M is obtained from the ground instantiation of Π by (i) deleting all rules having a negative literal C such that $M \models C$; (ii) deleting all negative literals form the remaining rules. ■

From a database instance we can define a structure.

Definition A.2. For a database instance D, $M_{ER}(D)$ is the Herbrand structure $\langle \mathcal{U}, I_{\mathcal{P}}, I_{\mathcal{B}} \rangle$, where $\mathcal{U}$ is the domain of the database and $I_{\mathcal{P}}$, $I_{\mathcal{B}}$ are the interpretations defined as follows:

1. If attributes values a_1 and a_2 match, then $Match(a_1, a_2) \in I_{\mathcal{P}}$.
2. If $R(\bar{s}) \in \bar{D}$, then $R(\bar{s}) \in I_{\mathcal{P}}$.
3. If $R(\bar{s}^1), R(\bar{s}^2) \in \bar{D}$, $\bar{s}^1 \neq \bar{s}^2$ and $\bar{s}^2 = \bar{s}^2 \cup \bar{s}^1$, then $Dominated(\bar{s}^1) \in I_{\mathcal{P}}$.
4. If $R(\bar{s}) \in I_{\mathcal{P}}$ and $Dominated(\bar{s}) \notin I_{\mathcal{P}}$, then $Er(\bar{s}) \in I_{\mathcal{P}}$

The interpretation for $I_{\mathcal{B}}$ is defined as expected: if Q is a built-in predicate, then $Q(a) \in I_{\mathcal{B}}$ iff $Q(a)$ is true in classical logic, and $Q(a) \notin I_{\mathcal{B}}$ iff $Q(a)$ is false. ■

Notice that the merge closure $\bar{D}$ is obtained using the match and merge functions in the union class for Swoosh and since we have a reflexive and commutative match function, the ICAR properties are satisfied by the match and merge functions. Then, merge domination order exists on tuples [Benjelloun et al. 2009].

Lemma A.3. $ER_{M_{ER}(D)} = \{Er(\bar{s}) \mid Er(\bar{s}) \in M_{ER}(D)\}$ is the unique entity resolution instance of D. ■

Proof: From [Benjelloun et al. 2009] we know that given match and merge functions that satisfy ICAR properties, any maximal derivation sequence starting from D computes entity resolution $ER(D)$. Therefore, we need to show that we can construct a maximal derivation sequence from $M_{ER}(D)$ that computes $ER(D)$ and $ER(D) = ER_{M_{ER}(D)}$ (since the match and merge functions in the union class stratify ICAR properties). Starting from D, perform all necessary merge steps to generate tuples in $ER(D)$. This is possible form $M_{ER}(D)$, since all the required tuples are in merge closure $\bar{D}$. Then, perform purge steps to remove all records which are not in $ER(D)$. Each step is a valid purge one, since $\bar{D} \leq ER(D)$. No additional purge steps are possible, since $ER(D)$ does not contain dominated tuples and no additional merge steps are possible, since $\bar{D} \leq ER(D)$. Therefore, the derivation is maximal, and it is clear that, according to construction of $M_{ER}(D)$, we have $ER(D) = ER_{M_{ER}(D)}$, since tuples in relation $Er(\bar{s})$ of $M_{ER}(D)$ are tuples of $\bar{D}$ such that are not dominated. ■

Lemma A.4. If D' is the unique entity resolution instance of D, then there is a model M of the program $\Pi^{UC}(D)$ such that $ER_M = D'$. Furthermore, $M_{ER}(D)$ is this model. ■

Proof: By Lemma A.3, we have that $ER_{M_{ER}(D)} = D'$, so we only need to show that $M_{ER}(D)$ satisfies all the rules of $(\Pi^{UC}(D))^M$. It is clear that, by construction of $M_{ER}(D)$, rules 1 and 4 are satisfied by $M_{ER}(D)$. There may exist a set of rules of form 2. If the body of the rule is satisfied, we have that $R(\bar{s}^1), R(\bar{s}^2) \in M_{ER}(D)$, $a_1 \in s_i^1$, $a_2 \in s_i^2$ such that a_1 and a_2 match. It means that for two tuples $r_1 = R(\bar{s}^1)$ and $r_2 = R(\bar{s}^2)$ we have $M(r_1, r_2) = true$ (since this rule exists when two tuples in $\Re$ match by having a pair of A_i values from them that match for some attributes A_i of R, $1 \leq i \leq n$). Therefore, merge closure of instance D should contain $R(\bar{s}^3)$ such that $\bar{s}^3 = \bar{s}^1 \cup \bar{s}^2$. Since in the construction of $M_{ER}(D)$, $R(\bar{s}) \in M_{ER}(D)$ iff $R(\bar{s}) \in \bar{D}$, the head of the rule is also satisfied and the whole rule is satisfied. Moreover, there is a set of rules of form 3. If the body of the rule is true in $M_{ER}(D)$, it means that $R(\bar{s}^1), R(\bar{s}^2) \in M_{ER}(D)$, $\bar{s}^2 = \bar{s}^1 \cup \bar{s}^2$, and for some s_i, $s_i^1 \neq s_i^2$. We know that $R(\bar{s}) \in M_{ER}(D)$ iff $R(\bar{s}) \in \bar{D}$. Therefore, $R(\bar{s}^1), R(\bar{s}^2) \in \bar{D}$. By construction of $M_{ER}(D)$ this implies that $Dominated(\bar{s}^1) \in M_{ER}(D)$ (since $\bar{s}^2 = \bar{s}^1 \cup \bar{s}^2$ and $\mu(r_1, r_2) = r_2$). Therefore, the rule is satisfied. ■

B DLV Code

Example 3:

```
%extensional database
r(t1,a1,b1).
r(t2,a2,b2).
r(t3,a3,b3).

%domain of database
dom(a1). dom(a2). dom(a3). dom(b1).
dom(b2). dom(b3). dom(b12). dom(b23).
dom(b123).

%existing similarities
att(a1,a2). att(b2,b3).
```

```
%matching functions
ma(b1,b2,b12). ma(b2,b3,b23).
ma(b1,b23,b123).

%rules related to match functions and
%similarity relations
attmatch(X,Y)  :- attmatch(Y,X).
ma(Y,X,Z)      :- ma(X,Y,Z).
attmatch(X,X)  :- dom(X).
ma(X,X,X)      :- dom(X).
ma(X,S,V)      :- ma(X,Y,Z), ma(W,Z,V),
                  ma(Y,W,S).
ma(Z,W,V)      :- ma(Y,W,S), ma(X,S,V),
                  ma(X,Y,Z).

%rules for obtaining clean solutions
match(T1,X1,Y1,T2,X2,Y2) v
   notmatch(T1,X1,Y1,T2,X2,Y2) :-
   r(T1,X1,Y1), r(T2,X2,Y2),
   att(X1,X2), Y1!=Y2, T1!=T2.
:- notmatch(T1,X1,Y1,T2,X2,Y2),
   not old(T1,X1,Y1),
   not old(T2,X2,Y2).
old(T1,X1,Y1):- r(T1,X1,Y1),
  r(T1,X1,Y2), ma(Y1,Y2,Y2), Y2!=Y1.
match(T2,X2,Y2,T1,X1,Y1) :-
        match(T1,X1,Y1,T2,X2,Y2).
r(T1,X1,Y3) :-
        match(T1,X1,Y1,T2,X2,Y2),
        ma(Y1,Y2,Y3).
match(T1,X1,Y1,T2,X2,Y2) v
   notmatch(T1,X1,Y1,T2,X2,Y2) :-
   r(T1,X1,Y1), r(T2,X2,Y2),
   att(Y1,Y2), Y1!=Y2,T1!=T2.
:- notmatch(T1,X1,Y1,T2,X2,Y2),
   not old(T1,X1,Y1),
   not old(T2,X2,Y2).
old(T1,X1,Y1)  :- r(T1,X1,Y1),
r(T1,X1,Y2), ma(Y1,Y2,Y2), Y2!=Y1.

prec(T1,X1,Y1,T2,X2,Y2,
     T1,X1,Y3,T4,X4,Y4) :-
     match(T1,X1,Y1,T2,X2,Y2),
     match(T1,X1,Y3,T4,X4,Y4),
     ma(Y1,Y3,Y3), Y3!=Y1.
prec(T1,X1,Y1,T2,X2,Y2,
     T1,X1,Y1,T4,X4,Y4) :-
     match(T1,X1,Y1,T2,X2,Y2),
     match(T1,X1,Y1,T4,X4,Y4),
     ma(Y1,Y4,Y3),Y1!=Y3.
prec(T1,X1,Y1,T2,X2,Y2,
     T1,X1,Y1,T2,X2,Y2) :-
     match(T1,X1,Y1,T2,X2,Y2).
:- prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,
        T1,X1,Y1,T2,X2,Y2),
        Y1 !=Y3.
:- prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,
        T1,X1,Y1,T2,X2,Y2),
        T2!=T4.
:- prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,
        T1,X1,Y1,T2,X2,Y2),
        X2 != X4.
:- prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,
        T1,X1,Y1,T2,X2,Y2),
        Y2 !=Y4.
:- prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,
        T1,X1,Y5,T6,X6,Y6),
   not prec(T1,X1,Y1,T2,X2,Y2,
        T1,X1,Y5,T6,X6,Y6).


rc(T1,X1,Y1) :- r(T1,X1,Y1),
                  not old(T1,X1,Y1).
```

Example 7: Since in this example we exploit built-in predicates and functions from library of the system, the logic program must contain, in the preamble, a line that tells the system to include the library itself. Other used predicates have the same meanings as mentioned before in the example.

```
#include<ListAndSet>
%extensional database
r({a1},{b1}). r({a2},{b2}).
r({a3},{b3}).

%existing similarities
match(a1,a2). match(a2,a3). match(a3,a2).
match(a2,a1).

%rules for obtaining resolution instance
% #Union is a predicate with three
% arguments saying that the third
% argument is the union of two other
% arguments. Another predicate is #Member
% saying that the first argument is a
% value in the second argument which is
% a set

r(As3,Bs3) :-     #union(As1,As2,As3),
                  #union(Bs1,Bs2,Bs3),
                  r(As1,Bs1),
                  r(As2,Bs2),
                  #member(S1,As1),
                  #member(S2,As2),
                  match(S1,S2).

dominated(As1,Bs1) :- r(As1,Bs1),
```

```
                    r(As2,Bs2),
                    #union(As1,As2,As2),
                    As2!= As1,
                    #union(Bs1,Bs2,Bs2).

dominated(As1,Bs1) :- r(As1,Bs1),
                    r(As2,Bs2),
                    #union(As1,As2,As2),
                    Bs2!= Bs1,
                    #union(Bs1,Bs2,Bs2).


er(As1,Bs1) :- r(As1,Bs1),
               not dominated(As1,Bs1).
```

C Cleaning programs for UC-Swoosh via MDs

For this end, we can consider possibly denumerable domain D_{A_i} for each attribute A_i of R, $1 \leq i \leq n$, and defines a reflexive and symmetric similarity relation $\approx_{A_i}$ for each domain. Then, the domain for each A_i becomes Dom_{A_i} such that it is the union of all subsets of D_{A_i}. Consequently, for each $R(\bar{s}) \in \mathfrak{R}$, s_i belongs to Dom_{A_i}. It also defines a similarity relation $\approx_{\{A_i\}}$ which can be induced from $\approx_{A_i}$, saying that $s_1 \approx_{\{A_i\}} s_2$ holds iff there exists $a_1 \in s_1$ and $a_2 \in s_2$ such that $a_1 \approx_{A_i} a_2$. Moreover, matching functions $m_{\{A_i\}}$ on $Dom_{A_i} \times Dom_{A_i}$ are considered such that $m_{\{A_i\}}(s_1, s_2) = s_1 \cup s_2$.

Based on these definitions, for union case of Swoosh given $r_1 = R(\bar{s}^1), r_2 = R(\bar{s}^2)$, $M(r_1, r_2)$ holds iff for some i, $s_i^1 \approx_{\{A_i\}} s_i^2$, and when $M(r_1, r_2) = true$, $\mu(r_1, r_2) = R(m_{\{A_1\}}(s_1^1, s_1^2), ..., m_{\{A_n\}}(s_n^1, s_n^2))$.

On the other hand, we can consider the following set of MDs Σ^S for $1 \leq i, j \leq n$ with $\approx_{\{A_i\}}$ and matching functions $m_{\{A_i\}}$ to reconstruct Swoosh framework under defined M and μ. Moreover, it is also assumed that tuples have tuple identifiers which are the first and extra attribute of relation R:

$$R[A_i] \approx_{\{A_i\}} R[A_i] \longrightarrow R[A_j] \doteq R[A_j]$$

Example 10. Consider the instance D^{ID} which differs from D in having tuple identifiers and the following set of MDs Σ^S:

$R(D^{ID})$	A	B
t_1	$\{a_1\}$	$\{b_1\}$
t_2	$\{a_2\}$	$\{b_2\}$
t_3	$\{a_3\}$	$\{b_3\}$

$$\phi_1 : R[A] \approx_{\{A\}} R[A] \longrightarrow R[B] \doteq R[B]$$
$$\phi_2 : R[A] \approx_{\{A\}} R[A] \longrightarrow R[A] \doteq R[A]$$

Assume $b_{12} \approx b_2$, $b_{12} \approx b_3$. Then, program $\Pi(D^{ID}, \Sigma^S)$ contains rules as follows:

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

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

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

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

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

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

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

   $R'(T_1, X_3, Y_1) \leftarrow Match_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_{\{A\}}(X_1, X_2, X_3).$

4. $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1', T_3, X_3, Y_3) \leftarrow Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_2}(T_1, X_1, Y_1', T_3, X_3, Y_3), Y_1 \preceq Y_1',\ Y_1 \neq Y_1'.$

   $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1', T_3, X_3, Y_3) \leftarrow Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_2}(T_1, X_1', Y_1, T_3, X_3, Y_3), X_1 \preceq X_1',\ X_1 \neq X_1'.$
   $(similarly\ for\ other\ mentioned\ cases)$

5. $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \leftarrow Match_{\phi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_1}(T_1, X_1, Y_1, T_3, X_3, Y_3), M_{\{B\}}(Y_1, Y_3, Y_4),\ Y_1 \neq Y_4.$

   $Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \leftarrow Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2), Match_{\phi_2}(T_1, X_1, Y_1, T_3, X_3, Y_3), M_{\{A\}}(X_1, X_3, X_4),\ X_1 \neq X_4.$
   $(similarly\ for\ other\ mentioned\ cases)$

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

7. $\leftarrow Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1', Y_1', T_3, X_3, Y_3), Prec(T_1, X_1', Y_1', T_3, X_3, Y_3, T_1, X_1, Y_1, T_2, X_2, Y_2), (T_1, X_1, Y_1, T_2, X_2, Y_2) \neq (T_1, X_1', Y_1', T_3, X_3, Y_3).$

8. $\leftarrow Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1', Y_1', T_3, X_3, Y_3), Prec(T_1, X_1', Y_1', T_3, X_3, Y_3, T_1, X_1'', Y_1'', T_4, X_4, Y_4), not\ Prec(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1'', Y_1'', T_4, X_4, Y_4).$

9. $R^c(T_1, X_1, Y_1) \leftarrow R'(T_1, X_1, Y_1), not\ OldVersion(T_1, X_1, Y_1).$

Then, it has one stable model which corresponds to the clean instance D^m :

$R(D^m)$	A	B
t_1	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$
t_2	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$
t_3	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$

■

As example 5 and 10 show in the union case of Swoosh the entity resolution instance is equivalent to the clean instance when we consider the instance containing tuples without tuple identifiers and eliminating duplicated tuples.

It can be proved that given a clean instance D^m obtained by enforcement of the above MDs Σ^S from D and the entity resolution instance D^s obtained directly via union case of Swoosh, (a) for every $R(\bar{s}) \in D^s$ there is a tuple $R(t, \bar{s}) \in D^m$; (b) for every tuple $R(t, \bar{s}) \in D^m$, there is $R(\bar{s}') \in D^s$ such that $R(\bar{s}) \leq R(\bar{s}')$. More precisely, union case of Swoosh entity resolution is equivalent to a clean instance resulting from a chase sequence with MDs when eliminating tuple identifiers there exist no dominated tuples in it [Bertossi, Kolahi and Lakshmanan 2011].

Proposition 8. Given an instance D_0, a set of MDs Σ^S, stable models S^s of program $\Pi^{UC}(D_0)$ and S^m of program $\Pi(D_0^{ID}, \Sigma^S)$, there is a one to one correspondence between clean instance D^m constructed from S^m and entity resolution instance D^s obtained from S^s. ■

To highlight the power of the proposed logic program for obtaining clean instances by enforcement of MDs we can also show that the logic program $\Pi^{UC}(D_0)$ for obtaining the entity resolution instance based on union case of Swoosh can be as the particular case of the general logic program $\Pi(D_0^{ID}, \Sigma)$ for obtaining clean instances (D_0 and D_0^{ID} just differ in having tuple identifiers). To obtain such logic programming specification for union case of Swoosh framework under defined M and μ we should consider the defined set of MDs Σ^S for $1 \leq i, j \leq n$ with $\approx_{\{A_i\}}$ and matching functions $m_{\{A_i\}}$. As mentioned earlier both programs try to merge tuples that match based on their approaches. In the case of $\Pi(D_0^{ID}, \Sigma^S)$ it does not allow old versions of tuples in future matches, therefore it uses program denial constraints which is what Swoosh doesn't require. Indeed, Swoosh does want to merge based on older version of tuples. Consequently, by syntactic modification of rules in $\Pi(D_0^{ID}, \Sigma^S)$ we can obtain $\Pi^{UC}(D_0)$. Particularly, we should modify and eliminate rules in $\Pi(D_0^{ID}, \Sigma^S)$ related to record the relative order of matching applications on each tuple identifier. Example 11 shows how the program $\Pi^{UC}(D)$ in example 5 can be obtained from $\Pi(D^{ID}, \Sigma^S)$ in example 10.

Example 11. In $\Pi(D^{ID}, \Sigma^S)$ by

- eliminating the denial constraint and rules for symmetry of $Match$ in 2
- modifying rules in 2 such that the relation $NotMaatch$ is eliminated from them
- eliminating rules related to the relation $Prec$
- renaming relations $OldVersion$ and R^c by $Dominated$ and Er_{MD} respectively, and
- combining the two following rules (similarly for $Match_{\phi_2}$):

$$Match_{\phi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx_{\{A\}} X_2,\ Y_1 \neq Y_2.$$

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

To get the rule:

$$R'(T_1, X_1, Y_3) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx_{\{A_i\}} X_2,\ Y_1 \neq Y_2, M_{\{B\}}(Y_1, Y_2, Y_3).$$

we can obtain the following program that differs form $\Pi^{UC}(D)$ in having tuple identifiers.

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

2. $R'(T_1, X_1, Y_3) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx_{\{A\}} X_2,\ Y_1 \neq Y_2, M_{\{B\}}(Y_1, Y_2, Y_3).$
$R'(T_1, X_3, Y_1) \leftarrow R'(T_1, X_1, Y_1),\ R'(T_2, X_2, Y_2),\ X_1 \approx_{\{A\}} X_2,\ X_1 \neq X_2, M_{\{A\}}(X_1, X_2, X_3).$

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

4. $Er_{MD}(T_1, X_1, Y_1) \leftarrow R'(T_1, X_1, Y_1), not\ Dominated(T_1, X_1, Y_1).$

In order to have a full correspondence, we have to add to the general cleaning program above the rule 5 for pruning dominated tuples. Consequently, relation Er_S collects the atoms of entity resolution instance.

5. $Er_S(X_1, Y_1) \leftarrow Er_{MD}(T_1, X_1, Y_1).$

■

As seen in the above example, the logic program $\Pi^{UC}(D)$ for obtaining entity resolution instances via union case of Swoosh can be obtained from the program $\Pi(D^{ID}, \Sigma^S)$ for obtaining clean instances w.r.t. MDs. Actually, $\Pi^{UC}(D)$ is a particular case of general program $\Pi(D^{ID}, \Sigma^S)$.

D ASPs for Swoosh with Negative Rules

In practice ER instances may contain inconsistencies. Actually, the process for matching and merging tuples is most often application-specific, complex, and error-prone and the initial database instance may contain ambiguous data which may be impossible to capture all the application nuances in ER process.

To remove the inconsistencies [Whang, Benjelloun and Garcia-Molina 2009] introduced *negative rules* that disallow inconsistencies in the ER instance(ER-N). Indeed, [Whang, Benjelloun and Garcia-Molina 2009] studied how to modify the ER process, in light of some integrity constraints called negative rules. Hence, a set of ER tuples satisfy the constraints. Furthermore, since in general there can be more than one valid ER instances, it was discussed how a domain expert can *guide* the ER process to capture a *desirable* and valid set of tuples using various methods for resolving tuples. Moreover, it was formally defined what is the valid ER instance in the presence of such negative rules: Given an instance D and the merge closure, $\bar{D}$, an *ER-N* of $\bar{D}$ is a consistent set of tuples D' that satisfies the conditions: (1) $D' \subseteq \bar{D}$. (2) $\forall$ r $\in \bar{D}$ - D': $\exists\ r' \in D'$ such that $r \preceq_s r'$ or $D' \cup \{r\}$ is inconsistent. (3) no strict subset of D' satisfies the first two conditions. (4) no other instances satisfying the first three conditions dominate D'.

Example 12. Consider the database instance D, shown blow, which are to be resolved (to represent tuples better, we are using r_i with $1 \leq i \leq 3$).

$R(D)$	$name$	$phone$	$gender$
r_1	$\{Mishael\}$	$\{7654321\}$	$\{\}$
r_2	$\{Michael\}$	$\{\}$	$\{M\}$
r_3	$\{Mishael\}$	$\{7654321\}$	$\{F\}$

Assume $M(r_1, r_2) = true$ and $M(r_1, r_3) = true$. Suppose first r_1 and r_2 are merged into new tuple r_{12}.

$R(D_1)$	$name$	$phone$	$gender$
r_{12}	$\{Mishael,\ Michael\}$	$\{7654321\}$	$\{M\}$
r_3	$\{Mishael\}$	$\{7654321\}$	$\{F\}$

Now suppose that $M(r_{12}, r_3) = true$ (since they have similar names). The result is new tuple r_{123}.

$R(D_2)$	$name$	$phone$	$gender$
r_{123}	$\{Mishael,\ Michael\}$	$\{7654321\}$	$\{F, M\}$

D_2 is the entity resolution instance for D. However, it is easy to see that there are *problems* with this instance which can be identified by negative rules. Consider we have a negative rule saying that one person cannot have two genders, and hence r_{123} variolates the constraint. To resolve the gender inconsistency, say we unmerge r_{123} back into $\{r_{12}, r_3\}$. Suppose we have a negative rule stating that no two tuples in ER should have the same phone number. So, the set $\{r_{12}, r_3\}$ is still not a valid ER instance. In this example, the problem occurred because r_1 was initially merged with r_2 instead of r_3. With the help of a domain expert, we can handle this situation. ■

[Whang, Benjelloun and Garcia-Molina 2009] followed an approach in which with the help of a domain expert, it is started identifying tuples that wanted to be in ER. Actually, the expert looks at the tuples, and selects one that is consistent, non dominated and *more desirable* to have in the final instance in order to prevent inconsistencies.

The proposed logic program for specification of union case for Swoosh can be extended such that it generates consistent ER instances in the presence of negative rules. Since for obtaining the *most desirable* consistent instances we need an expert to make decisions when there is a choice to be made, the program can have calls to external sources to handle it. HEX programs, which are non-monotonic logic programs admitting external atoms can be used for this purpose [Eiter et al. 2005]. By means of HEX programs, the new logic program delegates the task of identifying tuples that is *more desirable* to be in the ER instance to an external computational source (e.g., an external deduction system). So, the new logic program significantly differs from the one for union case of Swoosh without any negative rules. The main reason is that in ER with negative rules an expert chooses more desirable tuples to be in ER instance, and based on those tuples dominated and inconsistent tuples are eliminated not to be chosen as a tuple in ER. In contrast, in Swoosh we just focus on the merge closure and non dominated tuples in it are chosen to be in ER without any needs to look at which tuples are already in the ER instance.

Example 13. (Example 12 continued) We can have the following HEX program for obtaining ER in presence of the mentioned negative rules. As you can see the program is using higher order and external atoms to delegate the task of choosing more desirable tuples in ER. According to [Whang, Benjelloun and Garcia-Molina 2009] an expert chooses tuples from merge closure that wants to be in the ER instance and then those tuples that are dominated and inconsistent w.r.t. tuples in ER would be removed (for simplicity facts are not shown):

1. The following rule is related to generating merge closure of D:

$$R'(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2), \#Union(S_3^1, S_3^2)) \leftarrow R'(S_1^1, S_2^1, S_3^1),\ R'(S_1^2, S_2^2, S_3^2), \#Member(A_1, S_1^1), \#Member(A_2, S_1^2),\ Match_{\underline{A}}(A_1, A_2), (S_1^1, S_2^1, S_3^1) \neq (S_1^2, S_2^2, S_3^2).$$

2. This rule checks if a tuple is inconsistent in terms of having two genders:

$$Inconsistent(\bar{S}) \leftarrow R'(\bar{S}), \#Member(A_1, S_3), \\ \#Member(A_2, S_3),\ A_1 \neq A_2.$$

3. We need a rule to determine tuples of merge closure that are not dominated and inconsistent in order to be selected by an expert to be located in ER instances as more desirable ones:

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

4. The following rule computes the predicate *Er* taking values from the predicate *#External*, which chooses via *#External[Select]* more desirable tuples from tuples in *Select* to be located in *Er*, delegating this task to an external computational source to behave like a domain expert.

$$Er(\bar{S}) \leftarrow \#External[Select](\bar{S}).$$

5. We also have an inconsistency if there are two tuples with identical phone number. This rule determines those tuples that are inconsistent with tuples in ER and make them unavailable to be chosen as a tuple in ER.

$$Inconsistent(\bar{S}) \leftarrow Er(\bar{S^1}), R'(\bar{S^2}), \\ S_2^1 = S_2^2,\ \bar{S^1} \neq \bar{S^2}.$$

6. We need a rule to find dominated tuples w.r.t. tuples in ER.

$$Dominated(S_1^1, S_2^1, S_3^1) \leftarrow R'(S_1^1, S_2^1, S_3^1),\ Er(S_1^2, S_2^2, S_3^2), \\ (\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2), \#Union(S_3^1, S_3^2)) = \\ (S_1^2, S_2^2, S_3^2),\ (S_1^1, S_2^1, S_3^1) \neq (S_1^2, S_2^2, S_3^2).$$

■