



Consiglio Nazionale delle Ricerche

**On Temporal, Nonmonotonic, Nondeterministic Logic
Databases**

Fosca Giannotti, Giuseppe Manco
Mirco Nanni, Dino Pedreschi

Rapporto
CNUCE-B4-2000-016

CNUCE

Pisa

On Temporal, Nonmonotonic, Nondeterministic Logic Databases

Fosca Giannotti¹, Giuseppe Manco¹, Mirco Nanni² and Dino Pedreschi²

¹ CNUCE Institute of CNR, Via S. Maria 36, 56125 Pisa, Italy
email: F.Giannotti@cnuce.cnr.it, G.Manco@guest.cnuce.cnr.it

² Dipartimento di Informatica, Università di Pisa, C.so Italia 40, 56125 Pisa, Italy
email: {nnanni,pedre}@DI.Unipi.IT

Abstract. We consider in this paper an extension of Datalog with mechanisms for temporal, nonmonotonic and nondeterministic reasoning, which we refer to as Datalog++. We show, by means of examples, its flexibility in expressing queries concerning aggregates and data cube. Also, we show how iterated fixpoint and stable model semantics can be combined to the purpose of clarifying the semantics of Datalog++ programs, and supporting their efficient execution. Finally, we provide a more concrete implementation strategy, on which basis the design of optimization techniques tailored for Datalog++ is addressed.

Notes. This paper is a revised, extended version of two extended abstracts, appeared as [8, 9].

Keywords: Logic Programming, Databases, Negation, Nondeterminism, Stable models.
ACM-CR Subject Classification: D.1.6, I.2.3, H.2.3, F.1.3, F.3.2, F.3.3.

1 Introduction

Motivations. The name **Datalog++** is used in this paper to refer to Datalog extended with mechanisms supporting:

- *temporal* reasoning, by means of temporal, or *stage*, arguments of relations, ranging over a discrete temporal domain, in the style of Datalog_{IS} [4];
- *nonmonotonic* reasoning, by means of a form of stratified negation w.r.t. the stage arguments, called *XY-stratification* [23];
- *nondeterministic* reasoning, by means of the nondeterministic choice construct [11].

Datalog++, which is essentially a fragment of *LDL++* [2], and is advocated in [24, Chap. 10], revealed a highly expressive language, with applications in diverse areas such as AI planning [3], active databases [22], object databases [7], semistructured information management and Web restructuring [10]. However, a thorough study of the semantics of Datalog++ is still missing, which provides a basis to sound and efficient implementations and optimization techniques. A preliminary study of the semantics for a generalization of Datalog++ is sketched in [3], but their approach presents some inconveniences that are fixed in this paper.

Objective. The goal of this paper is to provide a declarative semantics for Datalog++, which accommodates and integrates the temporal, nonmonotonic and nondeterministic mechanisms, and which justifies the adoption of an iterated fixpoint semantics for the language, thus making it viable an efficient implementation.

We proceed as follows:

1. a natural, non effective, semantics for Datalog++ is assigned using the notion of a *stable model*;
2. an effective semantics is then assigned using an iterative procedure which exploits the stratification induced by the progression of the temporal argument;
3. in the main result of this paper, we show that 1. and 2. are equivalent, provided that a natural syntactic restriction is fulfilled, which imposes a disciplined use of the temporal argument within the *choice* construct.

On the basis of this result, we introduce a more concrete operational semantics, and a repertoire of optimization techniques, especially tailored for Datalog++. In particular, we discuss how it is possible to support efficient history-insensitive temporal reasoning by means of real side-effects during the iterated computation [16].

Related Work. Nondeterminism is introduced in deductive databases by means of the *choice* construct. The original proposal in [14] was later revised in [20], and refined in [11]. These studies exposed the close relationship connecting nonmonotonic reasoning with nondeterministic constructs, leading to the definition of a stable model semantics for *choice*. While the declarative semantics of *choice* is based on *stable model* semantics which is untractable in general, *choice* is amenable to efficient implementations, and it is actually supported in the logic database language *LDL* [17] and its evolution *LDL++* [2].

On the other side, stratification has been a crucial notion for the introduction of nonmonotonic reasoning in deductive databases. From the original idea in [1] of a static stratification based on predicate dependencies, stratified negation has been refined to deal with dynamic notions, as in the case of locally stratified programs [18] and modularly stratified programs [19]. Dynamic, or local, stratification has a close connection with temporal reasoning, as the progression of time points yields an obvious stratification of programs—consider for instance *Datalog_{IS}* [4]. It is therefore natural that non monotonic and temporal reasoning are combined in several deductive database languages, such as those in [15], [13], [10], [24, Chap. 10].

However, a striking mismatch is apparent between the above two lines of research: nondeterminism leads to a multiplicity of (stable) models, whereas stratification leads to a unique (perfect) model. So far, no comprehensive study has addressed the combination of the two lines, which occurs in Datalog++, and which requires the development of a *non deterministic iterated fixpoint* procedure. We notice however the mentioned exception of [3], where an approach to this problem is sketched with reference to locally stratified programs augmented with *choice*. In the present paper, we present instead a thorough treatment of Datalog++ programs, and repair an inconvenience of the approach in [3] concerning the incompleteness of the iterated fixpoint procedure.

2 Query Answering with Datalog++

Datalog, the basis of deductive databases, is essentially a friendly syntax to express relational queries, and to extend the query facilities of the relational calculus with recursion. Datalog's sim-

plicity in expressing complex queries impacted on the database technology, and nowadays recursive queries/views have become part of the SQL3 standard. Recursive queries find natural applications in all areas of information systems where computing transitive closures or traversals is an issue, such as in bill-of-materials queries, route or plan formation, graph traversals, and so on.

However, it is widely recognized that the expressiveness of Datalog's (recursive) rules is limited, and several extensions, along various directions, have been proposed. In this paper, we address in particular two such directions, namely nondeterministic and nonmonotonic reasoning, supported respectively by the choice construct and the notion of XY-stratification. We introduce these mechanisms by means of a few examples, which are meant to point out the enhanced query capabilities. The extended Datalog query language is referred to with the name **Datalog++**.

Nondeterministic choice. The choice construct is used to nondeterministically select subsets of answers to queries, which obey a specified FD constraint. For instance, the rule

$$\text{st_ad}(\text{St}, \text{Ad}) \leftarrow \text{major}(\text{St}, \text{Area}), \text{faculty}(\text{Ad}, \text{Area}), \text{choice}((\text{St}), (\text{Ad})).$$

assigns to each student a unique, arbitrary advisor from the same area, since the choice goal constrains the `st_ad` relation to obey the FD $(\text{St} \rightarrow \text{Ad})$. Therefore, if the base relation `major` is formed by the tuples $\{ \langle \text{smith}, \text{db} \rangle, \langle \text{gray}, \text{se} \rangle \}$ and the base relation `faculty` is formed by the tuples $\{ \langle \text{brown}, \text{db} \rangle, \langle \text{scott}, \text{db} \rangle, \langle \text{miller}, \text{se} \rangle \}$, then there are two possible outcomes for the query `st_ad(St, Ad)`: either $\{ \langle \text{smith}, \text{brown} \rangle, \langle \text{gray}, \text{miller} \rangle \}$ or $\{ \langle \text{smith}, \text{scott} \rangle, \langle \text{gray}, \text{miller} \rangle \}$. In practical systems, such as *LDL++*, one of these two solutions is computed and presented as a result.

Thus, a first use of choice is in computing nondeterministic, nonrecursive queries. However, choice can be combined with recursion, as in the following rules which compute an arbitrary ordering of a given relation `r`:

$$\begin{aligned} \text{ord_r}(\text{root}, \text{root}). \\ \text{ord_r}(X, Y) \leftarrow \text{ord_r}(_, X), r(Y), \text{choice}(X, Y), \text{choice}(Y, X). \end{aligned}$$

Here `root` is a fresh constant, conveniently used to simplify the program. If the base relation `r` is formed by k tuples, then there are $k!$ possible outcomes for the query `ord_r(X, Y)`, namely a set:

$$\{ \text{ord_r}(\text{root}, \text{root}), \text{ord_r}(\text{root}, t_1), \text{ord_r}(t_1, t_2), \dots, \text{ord_r}(t_{k-1}, t_k) \}$$

for each permutation $\{t_1, \dots, t_k\}$ of the tuples of `r`. Therefore, in each possible outcome of the mentioned query, the relation `ord_r` is a total ordering of the tuples of `r`. The double choice constraint in the recursive rule specifies that the successor and predecessor of each tuple of `r` is unique.

Interestingly, choice can be employed to compute new *deterministic* queries, which are inexpressible in Datalog, as well as in pure relational calculus. A remarkable example is the capability of expressing *aggregates*, as in the following program which computes the summation aggregate over a relation `r`, which uses an arbitrary ordering of `r` computed by `ord_r`:

$$\begin{aligned} \text{sum_r}(\text{root}, 0). \\ \text{sum_r}(Y, N) \leftarrow \text{sum_r}(X, M), \text{ord_r}(X, Y), N = M + Y. \\ \text{total_sum_r}(N) \leftarrow \text{sum_r}(X, N), \neg \text{ord_r}(X, _). \end{aligned}$$

Here, $\text{sum}_r(X, N)$ is used to accumulate in N the summation up to X , with respect to the order given by ord_r . Therefore, the total sum is reconstructed from $\text{sum}_r(X, N)$ when X is the last tuple in the order. Notice the use of (stratified) negation to the purpose of selecting the last tuple. In practical languages, such as $\mathcal{LDL}++$, some syntactic sugar for aggregation is used as an abbreviation of the above program [23]:

$$\text{total_sum}_r(\text{sum} \langle X \rangle) \leftarrow r(X).$$

On the basis of this simple example, more sophisticated forms of aggregation, such as *datacube* and other OLAP functions, can be built. As an example, consider a relation $\text{sales}(\text{Date}, \text{Department}, \text{Sale})$, and the problem of aggregating sales along the dimensions Date and Department . Three aggregation patterns are then possible, corresponding to the various facets of the datacube: $\langle \text{Date}, * \rangle$, $\langle *, \text{Department} \rangle$, $\langle *, * \rangle$. The former two patterns correspond to the aggregation of sales along a single dimension (respectively Department and Date), and can be obtained from the original relation by applying the method shown above. The latter pattern, then, can be obtained by recursively applying such method to one of the two patterns previously computed, in order to aggregate along the remaining dimension. In case of several dimensions along which to aggregate we can simply repeat the process, aggregating at each step along a new (i.e., still non-aggregated) dimension.

A thorough account on programming with nondeterminism in deductive databases can be found in [6, 12].

The semantics of choice is assigned using the so-called *stable model* semantics of Datalog \neg programs, a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [5]. To define the notion of a stable model we need to introduce a transformation H which, given an interpretation I , maps a Datalog \neg program P into a positive Datalog program $H(P, I)$:

$$H(P, I) = \{A \leftarrow B_1, \dots, B_n \mid A \leftarrow B_1, \dots, B_n, \neg C_1, \dots, \neg C_m \in \text{ground}(P) \wedge \{C_1, \dots, C_m\} \cap I = \emptyset\}$$

Next, we define:

$$S_P(I) = T_{H(P, I)} \uparrow \omega$$

Then, M is said to be a *stable model* of P if $S_P(M) = M$. In general, Datalog \neg programs may have zero, one or many stable models. The multiplicity of stable models can be exploited to give a declarative account of nondeterminism.

We can in fact define the *stable version* of a program P , $SV(P)$, to be the program transformation where all the references to the choice atom in a rule $r : H \leftarrow B, \text{choice}(X, Y)$ are replaced by the atom $\text{chosen}_r(X, Y)$, and define the chosen_r predicate with the following rules:

$$\begin{aligned} \text{chosen}_r(X, Y) &\leftarrow B, \neg \text{diffchoice}_r(X, Y). \\ \text{diffchoice}_r(X, Y) &\leftarrow \text{chosen}_r(X, Y'), Y \neq Y'. \end{aligned}$$

where, for any fixed value of X , each choice for Y inhibits all the other possible ones via diffchoice_r , so that in the stable models of $SV(P)$ there is (only) one of them. Notice that, by construction, each occurrence of a choice atom has its own pair of chosen and diffchoice atoms, thus bounding the scope of the atom to the rule it appears in. The various stable models of the transformed program $SV(P)$ thus correspond to the choice models of the original program.

XY-programs. Another notion used in this paper is that of XY-programs originally introduced in [23]. The language of such programs is $Datalog_{\neg, \exists}^{\perp}$, which admits negation on body atoms and a unary constructor symbol, used to represent a temporal argument usually called the *stage argument*. A general definition of XY-programs is the following. A set P of rules defining mutually recursive predicates, is an XY-program if it satisfies the following conditions:

1. each recursive predicate has a distinguished stage argument;
2. every recursive rule r is either an X-rule or a Y-rule, where:
 - r is an X-rule when the stage argument in every recursive predicates in r is the same variable,
 - r is a Y-rule when (i) the head of r has a stage argument $s(J)$, where J is a variable, (ii) some goal of r has J as its stage argument, and (iii) the remaining recursive goals have either J or $s(J)$ as their stage argument.

Intuitively, in the rules of XY-programs, an atom $p(J, _)$ denotes the extension of relation p at the current stage (present time) J , whereas an atom $p(s(J), _)$ denotes the extension of relation p at the next stage (future time) $s(J)$. By using a different *primed* predicate symbol p' in the $p(s(J), _)$ atoms, we obtain the so-called *primed version* of an XY-program. We say that an XY-program is *XY-stratified* if its primed version is a stratified program. Intuitively, if the dependency graph of the primed version has no cycles through negated edges, then it is possible to obtain an ordering on the original rules modulo the stage arguments. As a consequence, an XY-stratified program is also locally stratified, and has therefore a unique stable model that coincides with its perfect model [18].

Let P be an XY-stratified program. Then, for each $i > 0$, define P_i as

$$P_i = \{r[s^i(nil)/I] \mid r \in P, I \text{ is the stage argument of the head of } r\}$$

(here $r[x/I]$ stands for r where I is replaced by x) i.e., P_i is the set of rule instances of P that define the predicates with stage argument $s^i(nil) = i$. Then the iterated fixpoint procedure for computing the (unique) minimal model of P can be defined as follows:

1. compute M_0 as the minimal model of P_0 ;
2. for each $j > 0$ compute M_j as the minimal model of $P_j \cup M_{j-1}$.

Notice that for each $j \geq 0$, P_j is stratified by the definition, and hence its perfect model M_j is computable via an iterated fixpoint procedure.

In this paper, we use the name $Datalog_{++}$ to refer to the language of XY-programs augmented with choice goals.

3 A Semantics for $Datalog_{++}$

When choice constructs are allowed in XY-programs, a multiplicity of stable models exists for any given program, and therefore it is needed to clarify how this phenomenon combines with the iterated fixpoint semantics of choice-free XY-programs. This task is accomplished in three steps.

1. First, we present a general result stating that, whenever a $Datalog_{\neg}$ program P is stratifiable into a hierarchy of recursive cliques Q_1, Q_2, \dots , then any stable model of the entire program P can be reconstructed by iterating the construction of approximating stable models, each associated to a clique.

2. Second, we observe that, under a syntactic restriction on the use of the choice construct that does not compromise expressiveness, Datalog++ programs can be naturally stratified into a hierarchy of recursive cliques Q_1, Q_2, \dots , by using the temporal arguments of recursive predicates.
3. Third, by the observation in 2., we can apply the general result in 1. to Datalog++ programs, thus obtaining that the stable models of the entire program can be computed by an iterative fixpoint procedure which follows the stratification induced by the temporal arguments.

Given a (possibly infinite) program P , consider a (possibly infinite) topological sort of its distinct recursive cliques $Q_1 \prec Q_2 \prec \dots \prec Q_i \prec \dots$, induced by the dependency relation over the predicates of P . Given an interpretation I , we use the notation I_i to denote the subset of atoms of I whose predicate symbols are predicates defined in clique Q_i .

The following observations are straightforward:

- $\bigcup_{i>0} I_i = I$, and analogously $\bigcup_{i>0} Q_i = P$;
- the predicates defined in Q_{i+1} depend only on the definitions in $Q_1 \cup \dots \cup Q_i$; as a consequence, the interpretation of Q_{i+1} is $I_1 \cup \dots \cup I_i \cup I_{i+1}$ (i.e., we can ignore $\bigcup_{j>i+1} I_j$).

The next definition shows how to transform each clique, within the given topological ordering, in a self-contained program which takes into account the information deduced by the previous cliques. Such transformation resembles the Gelfond-Lifschitz transformation reported in Sect. 2.

Definition 1. Consider a program P , a topological sort of its cliques $Q_1 \prec Q_2 \prec \dots \prec Q_i \prec \dots$, and an interpretation $I = \bigcup_{i>0} I_i$. Now define

$$\begin{aligned}
Q_i^{red(I)} = \{ & H \leftarrow B_1, \dots, B_n \mid H \leftarrow B_1, \dots, B_n, C_1, \dots, C_m \in \text{ground}(Q_i) \\
& \wedge B_1, \dots, B_n \text{ are defined in } Q_i \\
& \wedge C_1, \dots, C_m \text{ are defined in } (Q_1 \cup \dots \cup Q_{i-1}) \\
& \wedge I_1 \cup \dots \cup I_{i-1} \models C_1, \dots, C_m \}
\end{aligned}$$

□

The idea underlying the transformation is to remove from each clique Q_i all the dependencies induced by the predicates which are defined in lower cliques. We abbreviate $Q_i^{red(I)}$ by Q_i^{red} , when the interpretation I is clear by the context.

Example 1. Consider the program $P = \{p \leftarrow q, r. \quad q \leftarrow r, t. \quad r \leftarrow q, s.\}$ and the cliques $Q_1 = \{q \leftarrow r, t. \quad r \leftarrow q, s.\}$ and $Q_2 = \{p \leftarrow q, r.\}$. Now, consider the interpretation $I = \{s, q, r\}$. Then $Q_1^{red} = \{q \leftarrow r, t. \quad r \leftarrow q, s.\}$ and $Q_2^{red} = \{p \leftarrow .\}$. □

The following Lemma 1 states the relation between the models of the transformed cliques and the models of the program. We abbreviate $I_1 \cup \dots \cup I_i$ with $I^{(i)}$, and analogously for $Q^{(i)}$.

Lemma 1. Given a (possibly infinite) Datalog⁻ program P and an interpretation I , let $Q_1 \prec Q_2 \prec \dots \prec Q_i \prec \dots$ and $I_1 \prec I_2 \prec \dots \prec I_i \prec \dots$ be the topological sorts on P and I induced by the dependency relation of P . Then the following statements are equivalent:

1. $S_P(I) = I$
2. $\forall i > 0. S_{Q_i^{red}}(I_i) = I_i$
3. $\forall i > 0. S_{Q^{(i)}}(I^{(i)}) = I^{(i)}$

Proof sketch. The proof is structured as follows: (1) \iff (3) and (2) \iff (3).

(3) \implies (1) We next show that (a) $S_P(I) \subseteq I$, and (b) $I \subseteq S_P(I)$.

(a) Each rule in $H(P, I)$ comes from a rule r of P , which in turn appears in $Q^{(i)}$ for some i , and then $I^{(i)}$ is a model of r , by the hypothesis. No atom in $I \setminus I^{(i)}$ appears in r , so also I is model of r . I is then a model of $H(P, I)$, and hence $S_P(I) \subseteq I$.

(b) If $A \in I$, then $A \in I^{(i)}$ for some i , so (by the hypothesis and definition of S_P) for each I^* such that $I^* = T_{H(Q^{(i)}, I^{(i)})}(I^*)$, $A \in I^*$. Moreover, for each I' such that $I' = T_{H(P, I)}(I')$, it is readily checked that for each i $I'^{(i)} = T_{H(Q^{(i)}, I^{(i)})}(I'^{(i)})$, and then $I \subseteq S_P(I)$.

(1) \implies (3) We observe that $I = \min\{I^* \mid I^* = T_{H(P, I)}(I^*)\}$, which implies:

$$I^{(i)} = \min\{I^{*(i)} \mid I^{*(i)} = T_{H(Q^{(i)}, I^{(i)})}(I^{*(i)})\}.$$

(2) \implies (3) We proceed by induction on i . The base case is trivial. In the inductive case, we next show that (a) $S_{Q^{(i)}}(I^{(i)}) \subseteq I^{(i)}$, and (b) vice versa.

(a) Notice that from the induction hypothesis, $I^{(i)} \models Q^{(i-1)}$, and then it suffices to show that $I^{(i)} \models Q_i$ (by a simple case analysis).

(b) Exploiting the induction hypothesis, $I^{(i-1)} \subseteq S_{Q^{(i-1)}}(I^{(i-1)}) = S_{Q^{(i-1)}}(I^{(i)}) \subseteq S_{Q^{(i)}}(I^{(i)})$ (by definition of $H(P, I)$). We now show by induction on n that $\forall n \geq 0$ $T_{H(Q_i^{red}, I_i)}^n \subseteq T_{H(Q^{(i)}, I^{(i)})}^\omega$. The base case $n = 0$ is trivial. In the induction case ($n > 0$), if $A \in T_{H(Q_i^{red}, I_i)}^n$, then there exists a rule $A \leftarrow b_1, \dots, b_h$ in $H(Q_i^{red}, I_i)$ such that $\{b_1, \dots, b_h\} \subseteq T_{H(Q_i^{red}, I_i)}^{n-1}$.

Now, by definition of H and Q_i^{red} , there exists a rule:

$$A \leftarrow b_1, \dots, b_h, \neg c_1, \dots, \neg c_j, d_1, \dots, d_k, \neg e_1, \dots, \neg e_l$$

in Q_i such that $\{c_1, \dots, c_j\} \cap I_i = \emptyset$ and $I^{(i-1)} \models d_1 \wedge \dots \wedge d_k \wedge \neg e_1 \wedge \dots \wedge \neg e_l$. Observe now that by definition of H , $A \leftarrow b_1, \dots, b_h, d_1, \dots, d_k \in H(Q^{(i)}, I^{(i)})$. Furthermore, by the induction hypothesis and $I^{(i-1)} \subseteq S_{Q^{(i)}}(I^{(i)})$, we have the following: $\{b_1, \dots, b_h, d_1, \dots, d_k\} \subseteq T_{H(Q^{(i)}, I^{(i)})}^\omega$. Hence, by definition of T^ω , $A \in T_{H(Q^{(i)}, I^{(i)})}^\omega$, that is $A \in S_{Q^{(i)}}(I^{(i)})$. This completes the innermost induction, and we obtain that $I_i = S_{Q_i^{red}}(I_i) \subseteq S_{Q^{(i)}}(I^{(i)})$.

(3) \implies (2) We proceed in a way similar to the preceding case. To see that $\forall i$ $I_i \subseteq S_{Q_i^{red}}(I_i)$, it suffices to verify that for each rule instance r with head A , the following property holds: $\forall n$ $A \in T_{H(Q^{(i)}, I^{(i)})}^n \implies A \in T_{H(Q_i^{red}, I_i)}^n$. For the converse, we simply observe that I_i is a model of Q_i^{red} . \square

This result states that an arbitrary Datalog \neg program has a stable model if and only if each its approximating clique, according to the given topological sort, has a *local* stable model. This result gives us an intuitive idea for computing the stable models of an approximable program by means of the computation of the stable models of its approximating cliques.

Notice that Lemma 1 holds for arbitrary programs, provided that a stratification into a hierarchy of cliques is given. In this sense, this result is more widely applicable than the various notions of stratified programs, such as that of *modularly stratified programs* [19], in which it is required that each clique Q_i^{red} is locally stratified. On the contrary, we do not require here that each clique is, in any sense, stratified. This is motivated by the objective of dealing with non determinism, and justifies why we adopt the (nondeterministic) stable model semantics, rather than other deterministic semantics for (stratified) Datalog \neg programs, such as, for instance, perfect model semantics [18].

We turn now our attention to XY-programs. The result of instantiating the clauses of an XY-program P with all possible values (natural numbers) of the stage argument, yields a new program $SG(P)$ (for *stage ground*). More precisely, $SG(P) = \bigcup_{i \geq 0} P_i$, where

$$P_i = \{r[i/I] \mid r \text{ is a rule of } P, I \text{ is the stage argument of } r\}.$$

The stable models of P and $SG(P)$ are closely related:

Lemma 2. *Let P be an XY-program. Then, for each interpretation I :*

$$S_P(I) = I \iff S_{SG(P)}(I) = I$$

Proof sketch. We show by induction that $\forall n. T_{H(SG(P), I)}^n(\emptyset) = T_{H(P, I)}^n(\emptyset)$, which implies the thesis. The base case is trivial. For the inductive case, observe that since P is XY-stratified, if $A \in T_{H(P, I)}^{n+1}(\emptyset)$ then for each rule $A \leftarrow B_1, \dots, B_n \in H(P, I)$ such that $\{B_1, \dots, B_n\} \in T_{H(P, I)}^n(\emptyset) = T_{H(SG(P), I)}^n(\emptyset)$, we have $A \leftarrow B_1, \dots, B_n \in H(SG(P), I)$.

Vice versa, if $A \in T_{H(SG(P), I)}^{n+1}(\emptyset)$ then for each rule $A \leftarrow B_1, \dots, B_n \in H(SG(P), I)$ such that $\{B_1, \dots, B_n\} \in T_{H(SG(P), I)}^n(\emptyset) = T_{H(P, I)}^n(\emptyset)$, we have $A \leftarrow B_1, \dots, B_n \in H(P, I)$. \square

However, the dependency graph of $SG(P)$ (which is obviously the same as P) does not induce necessarily a topological sort, because in general XY-programs are not stratified, and therefore Lemma 1 is not directly applicable. To tackle this problem, we distinguish the predicate symbol p in the program fragment P_i from the same predicate symbol in all other fragments P_j with $j \neq i$, by differentiating the predicate symbols using the temporal argument. Therefore, if $p(i, x)$ is an atom involved in some rule of P_i , its modified version is $p_i(x)$. More precisely, we introduce, for any XY-program P , its modified version $SO(P)$ (for *stage-out*), defined by $SO(P) = \bigcup_{i \geq 0} SO(P)_i$ where $SO(P)_i$ is obtained from the program fragment P_i of $SG(P)$ by extracting the stage arguments from any atom, and adding it to the predicate symbol of the atom. Similarly, the modified version $SO(I)$ of an interpretation I is defined. Therefore, the atom $p(i, x)$ is in I iff the atom $p_i(x)$ is in $SO(I)$, where i is the value in the stage argument position of relation p .

Unsurprisingly, the stable models of $SG(P)$ and $SO(P)$ are closely related:

Lemma 3. *Let P be an XY-program. Then, for each interpretation I :*

$$S_{SG(P)}(I) = I \iff S_{SO(P)}(SO(I)) = SO(I).$$

Proof sketch. It is easy to see that $SO(SG(P)) = SO(P)$. Hence, the least Herbrand models of $SO(H(SG(P), I))$ and $H(SO(P), SO(I))$ coincide. \square

Our aim is now to conclude that, for a given Datalog++ program P :

- (a) $SO(P)_0 \prec SP(P)_1 \prec \dots$ is the topological sort over $SO(P)$ in the hypothesis of Lemma 1¹; recall that, for $i \geq 0$, the clique $SO(P)_i$ consists of the rules from $SO(P)$ with stage argument i in their heads;

¹ In general, $SO(P)_i$ can be composed by more than one clique, so that in the above expression it should be replaced by $SO(P)_i^1 \prec \dots \prec SO(P)_i^{n_i}$. However, for ease of presentation we ignore it, since such general case is trivially deducible from what follows.

(b) by Lemmas 1, 2 and 3, an interpretation I is a stable model of P iff I can be constructed as $\bigcup_{i \geq 0} I_i$, where, for $i \geq 0$, I_i is a stable model of $SO(P)_i^{red(I^{(i)})}$, i.e. the clique $SO(P)_i$ reduced by substituting the atoms deduced at stages earlier than i .

On the basis of (b) above, it is possible to define an iterative procedure to construct an arbitrary stable model M of P as the union of the interpretations M_0, M_1, \dots defined as follows:

Iterated stable model procedure.

Base case. M_0 is a stable model of the bottom clique $SO(P)_0$.

Induction case. For $i > 0$, M_i is a stable model of $SO(P)_i^{red(M^{(i)})}$, i.e. the clique $SO(P)_i$ reduced with respect to $M_0 \cup \dots \cup M_{i-1}$. \square

The interpretation $M = \bigcup_{i \geq 0} M_i$ is called an *iterated stable model* of P .

It should be observed that this construction is close to the procedure called *iterated choice fixpoint* in [3]. Also, following the approach of [12], each local stable model M_i can in turn be efficiently constructed by a nondeterministic fixpoint computation, in polynomial time.

Unfortunately, the desired result that the notions of stable model and iterated stable model coincide does not hold in full generality, in the sense that the iterative procedure is not complete for arbitrary Datalog++ programs. In fact, as demonstrated by the example below, an undisciplined use of choice in Datalog++ programs may cause the presence of stable models that cannot be computed incrementally over the hierarchy of cliques.

Example 2. Consider the following simple Datalog++ program P :

```

q(0, a).
q(s(I), b) ← q(I, a).
p(I, X) ← q(I, X), choice((), X).

```

In the stable version $SV(P)$ of P , the rule defining predicate p is replaced by:

```

p(I, X) ← q(I, X), chosen(X).
chosen(X) ← q(I, X), ¬diffchoice(X).
diffchoice(X) ← chosen(Y), Y ≠ X.

```

It is readily checked that $SV(P)$ admits two stable models, namely $\{q(0, a), q(s(0), b), p(0, a)\}$ and $\{q(0, a), q(s(0), b), p(s(0), b)\}$, but only the first model is an iterated stable model, and therefore the second model cannot be computed using the *iterated choice fixpoint* of [3]. \square

The technical reason for this problem is that the free use of the choice construct inhibits the possibility of defining a topological sort on $SO(P)$ based on the value of the stage argument. In the Example 2, the predicate dependency relation of $SO(SV(P))$ induces a dependency among stage i and the stages $j > i$, because of the dependency of the chosen predicate from the predicates q_i for all stages $i \geq 0$.

To prevent this problem, it suffices to require that choice goals refer the stage argument I in the domain of the associated functional dependency. The Datalog++ programs which comply with this constraint are called *choice-safe*. The following is a way to turn the program of Example 2 into a choice-safe program (with a different semantics):

$p(I, X) \leftarrow q(I, X), \text{choice}(I, X).$

This syntactic restriction, moreover, does not greatly compromise the expressiveness of the query language, in that it is possible to simulate within this restriction most of the general use of choice (see [16]).

The above considerations are summarized in the following main result of the paper, which, under the mentioned restriction of choice-safety, is a direct consequence of Lemmas 1, 2 and 3.

Theorem 1 (Correctness and completeness of the iterated stable model procedure).
Let P be a choice-safe Datalog++ program and I an interpretation. Then I is a stable model of $SV(P)$ iff it is an iterated stable model of P . \square

The following example shows a computation with the iterated stable model procedure.

Example 3. Consider the following Datalog++ version of the *seminative* program, discussed in [23], which non-deterministically computes a maximal path from node a over a graph g :

```

delta(0, a).
delta(s(I), Y) ← delta(I, X), g(X, Y), ¬all(I, Y), choice((I, X), Y).
all(I, X) ← delta(I, X).
all(s(I), X) ← all(I, X), delta(s(I), _).

```

Assume that the graph is given by $g = \{\langle a, b \rangle, \langle b, c \rangle, \langle b, d \rangle, \langle d, e \rangle\}$. The following interpretations are carried out at each stage of the iterated stable model procedure:

1. $I_0 = \{\text{delta}_0(a), \text{all}_0(a)\}.$
2. $I_1 = \{\text{all}_1(a), \text{all}_1(b), \text{delta}_1(b)\}.$
3. $I_2^1 = \{\text{all}_2(a), \text{all}_2(b), \text{delta}_2(c), \text{all}_2(c)\},$
 $I_2^2 = \{\text{all}_2(a), \text{all}_2(b), \text{delta}_2(d), \text{all}_2(d)\}$
4. $I_3^1 = \emptyset, I_3^2 = \{\text{all}_3(a), \text{all}_3(b), \text{all}_3(d), \text{delta}_3(e), \text{all}_3(e)\}$
5. $I_j = \emptyset$ for $j > 3$.

By Theorem 1, we conclude that there are two stable models for the program: $I^1 = I_0 \cup I_1 \cup I_2^1$ and $I^2 = I_0 \cup I_1 \cup I_2^2 \cup I_3^2$. Clearly, any realistic implementation, such as that provided in $\mathcal{LDL}++$, computes non deterministically only *one* of the possible stable models. \square

4 An Operational Semantics for Datalog++

We now translate the iterated stable model procedure into a more concrete form, by using relational algebra operations and control constructs. Following the style of [21], we associate with each predicate p a relation P — same name capitalized.

The elementary deduction step $T_Q(I)$ is translated as an assignment to appropriate relations:

$$I' := T_Q(I) \quad \longleftrightarrow \quad \forall p \in \text{def}(Q) \ P := \text{EVAL}(p, \text{Rels})$$

where $\text{Rels} = \{P \mid p \in \text{def}(Q)\} \cup \{R \mid R \in \text{EDB}\}$, i.e., the relations defined in the clique Q together with the extensional relations, and $\text{EVAL}(p, \text{Rels})$ denotes, in the notation of [21], a

single evaluation step of the rules for predicate p with respect to the current extension of relations in $Rels$.

We show the translation of Datalog++ cliques incrementally in three steps, starting with simple Datalog programs and stratified negation, then introducing the Choice construct and eventually describing how to translate the full language. The translation of a whole program can be trivially obtained by gathering the single translated cliques in the natural order.

Datalog with stratified negation. We can apply straightforwardly the transformation given in [21] for safe stratified Datalog programs, where each negative literal referring to a previously computed or extensional relation is translated to the complement of the relation w.r.t. the universe of constants:

Translation Template 1

$$\begin{array}{l} \forall p \text{ defined in } Q : P := \emptyset; \\ \text{repeat} \\ \quad \forall p \text{ defined in } Q : \text{last_}P := P; \\ \quad \quad P := \text{EVAL}(p, Rels); \\ \text{until } \forall p \text{ defined in } Q : P = \text{last_}P \end{array}$$

The translation is illustrated in the following:

Example 4. The program:

$$\begin{array}{l} p(\text{nil}, a). \\ p(X, Y) \leftarrow p(X, Z), \text{arc}(Z, Y), \neg \text{bad_node}(Y). \end{array}$$

is translated to the following naive evaluation procedure, where $\text{EVAL}(\dots)$ is instanced with an appropriate RA query:

$$\begin{array}{l} P := \emptyset; \\ \text{repeat} \\ \quad \text{last_}P := P; \\ \quad P := \{ \langle \text{nil}, a \rangle \} \cup \pi_{X,Y} \left(\text{last_}P(X, Z) \bowtie \text{Arc}(Z, Y) \bowtie \overline{\text{Bad_node}(Y)} \right); \\ \text{until } P = \text{last_}P. \end{array}$$

Adding Choice. Now we need to translate in relational algebra terms the operations which compose a nondeterministic computation. Following the approach of [12, 11], we partition the rules of $SV(Q)$ (the stable version of Q) into three sets:

$$\begin{array}{l} \mathcal{C} = \text{chosen rules} \\ \mathcal{D} = \text{diffChoice rules} \\ \mathcal{O} = SV(Q) \setminus (\mathcal{C} \cup \mathcal{D}) \text{ (i.e., the remaining rules)} \end{array}$$

Now, the non-deterministic fixpoint procedure which computes the stable models of a choice program is represented by the following:

Translation Template 2

0. **Init**
 $\forall p$ defined in $SV(Q) : P := \emptyset$
1. **Saturation**
repeat
 $\forall p$ defined in $\mathcal{O} : last_P := P$
 $P := EVAL(p, Rels)$
until $\forall p$ defined in $\mathcal{O} : last_P = P$
2. **Gather choices**
 $\forall chosen_r$ defined in $\mathcal{C} : Chosen'_r := EVAL(chosen_r, Rels)$
3. **Termination test**
if $\forall chosen_r$ defined in $\mathcal{C} : Chosen'_r = \emptyset$ then stop
4. **Choice**
Execute (fairly) the following
 - a. Choose $Chosen'_r \neq \emptyset$
 - b. Choose $\bar{t} \in Chosen'_r$
 - c. $Chosen_r := Chosen_r \cup \{\bar{t}\}$
5. **Inhibit other choices**
 $\forall diffchoice_r$ defined in $\mathcal{D} : Diffchoice_r := EVAL(diffchoice_r, Rels)$
6. goto 1

At step 1, the procedure tries to derive all possible atoms from the already given choices (none at the first iteration), so that at step 2 it can collect all candidate atoms which can be chosen later. If there is any such atom (i.e., we have not reached the fixpoint of the evaluation), then we can nondeterministically choose one of them (step 4) and then propagate the effects of such choice (step 5) in order to force the FD which it implies. We are then ready to repeat the process.

Example 5. The stable version of the students-advisors example seen in section 2 is the following:

- $$\begin{aligned} \mathcal{O} &: st_ad(St, Ad) \leftarrow major(St, Area), faculty(Ad, Area), chosen(St, Ad). \\ \mathcal{C} &: chosen(St, Ad) \leftarrow major(St, Area), faculty(Ad, Area), \neg diffchoice(St, Ad). \\ \mathcal{D} &: diffchoice(St, Ad) \leftarrow chosen(St, Ad'), Ad \neq Ad'. \end{aligned}$$

Following the above translation schema, then, we obtain the following procedure:

0. $St_ad := \emptyset; Chosen := \emptyset; Diffchoice := \emptyset;$
1. $St_ad := \pi_{St, Ad} (Major(St, Area) \bowtie Faculty(Ad, Area) \bowtie Chosen(St, Ad));$
2. $Chosen' := \pi_{St, Ad} (Major(St, Area) \bowtie Faculty(Ad, Area) \bowtie \overline{Diffchoice(St, Ad)});$
3. if $Chosen' = \emptyset$ then stop;
- 4b. Choose (fairly) $\langle st, ad \rangle \in Chosen'$;
- 4c. $Chosen := Chosen \cup \{\langle st, ad \rangle\};$
5. $Diffchoice := Diffchoice \cup (\{\langle st \rangle\} \times \{\langle ad \rangle\});$
6. goto 1

Notice that some steps have been slightly modified: (i) step 1 has been simplified, since the only rule in \mathcal{O} is not recursive (modulo the chosen predicate); (ii) here we have only one choice rule, so

step 4a becomes useless and then it has been ignored; (iii) step 5 is rewritten in a brief and more readable form, which has exactly the same meaning of that shown in the above general schema.

Adding XY-stratification. Analyzing the evaluation procedure of an XY-cliques Q , it is easy to see that at each step n the only atoms which can be derived are of the form $p(n, \dots)$, i.e., all with the same stage argument, and then the syntactic form of the rules ensures that such rules refer only to atoms $p(n, \dots)$ and $p(n-1, \dots)$. Then, the stage arguments in each rule serve only to distinguish the literals computed in the actual stage I from those computed in the previous stage $I-1$.

Therefore, we can safely omit the stage argument from each XY-recursive predicate, renaming the literals referring to a previous stage (i.e., those having stage I inside a rule with head having stage $I+1$) by adding the prefix "old.". This does not apply to exit-rules, in which the stage argument value is significant and then must be preserved. We denote by Q' the resulting rules, and by p' the predicate obtained from each p .

Example 6. The program Q :

$$\begin{aligned} & p(7, a). \\ & p(s(I), X) \leftarrow q(s(I), X, Y), r(I, X). \end{aligned}$$

is translated into the new program Q' :

$$\begin{aligned} & p(7, a). \\ & p(X) \leftarrow q(X, Y), \text{old}_r(X). \end{aligned}$$

Now, it suffices to store in an external register J the value of the stage under evaluation. We can (i) fire the exit-rules having the same stage argument as J , and then (ii) to evaluate the new rules in Q' (which are now stratified and possibly with choice) as described in the last two sections. When we have completely evaluated the actual stage, we need to store the newly derived atoms p in the corresponding old_p , to increment J and then to repeat the process in order to evaluate the next stage. The resulting procedure is the following.

Translation Template 3

0. Init

$$\begin{aligned} & J := 0; \\ & \forall p' \text{ defined in } Q' : \text{old}_P' := \emptyset \\ & P' := \emptyset \end{aligned}$$

1. Fire exit rules

$$\begin{aligned} & \forall \text{exit-rule } r = p(j, \dots) \leftarrow \dots : \\ & \quad \text{if } J = j \text{ then } P' := \text{EVAL}_r(p', \text{Rels}) \end{aligned}$$

2. Fire Q'

Translate Q' following the translation templates 1 and 2

3. Update relations

$$\begin{aligned} & \forall p' \text{ defined in } Q' : \text{old}_P' := P' \\ & P := P \cup \{ \{J\} \} \times P'; \\ & P' := \emptyset \end{aligned}$$

4. $J := J + 1$; goto 1

Here we simply reduce the evaluation of the XY-clique to the iterated evaluation of its *stage instances* (step 2) in a sequential ascending order (step 4). Each stage instance is stratified modulo choice and then it can be broken into subcliques (step 2) which can be translated by template 1 (if choice-free) or template 2 (if with choice). The resulting relations (step 3) can be easily obtained by collecting at each stage J the relations P^J and translating them into the corresponding P , i.e. adding to them the stage argument J .

Example 7. Let $g/2$ be an extensional predicate representing the edges of a graph. Consider the following clique Q :

$$\begin{aligned} \Delta(\text{nil}, Y) &\leftarrow g(a, Y). \\ \Delta(s(I), Y) &\leftarrow \Delta(I, X), g(X, Y), \neg \text{all}(I, Y). \\ \text{all}(s(I), Y) &\leftarrow \text{all}(I, Y), \Delta(s(I), -). \\ \text{all}(I, Y) &\leftarrow \Delta(I, Y). \end{aligned}$$

The corresponding transformed clique Q' is:

$$\begin{aligned} r_0 : \Delta(\text{nil}, Y) &\leftarrow g(a, Y). \quad (\text{exit rule}) \\ r_1 : \Delta'(Y) &\leftarrow \text{old}_\Delta(X), g(X, Y), \neg \text{old_all}'(Y). \\ r_2 : \text{all}'(Y) &\leftarrow \text{old_all}'(Y), \Delta'(-). \\ r_3 : \text{all}'(Y) &\leftarrow \Delta'(Y). \end{aligned}$$

Q' can be partitioned into: *exit-rule* r_0 , subclique $Q'_1 = \{r_1\}$ and subclique $Q'_2 = \{r_2, r_3\}$. Applying the translation template 3 we obtain:

0. $J := 0$; $\text{old}_\Delta := \emptyset$; $\Delta' := \emptyset$;
 $\text{old_All}' := \emptyset$; $\text{All}' := \emptyset$;
1. **if** $J = 0$ **then** $\Delta' := \pi_Y(\sigma_{X=a}(G(X, Y)))$;
- 2a. $\Delta' := \Delta' \cup \pi_Y(\text{old}_\Delta(X) \bowtie G(X, Y) \bowtie \text{All}'(Y))$;
- 2b. $\text{All}' := \text{All}' \cup \{\pi_Y(\text{old_All}'(Y) \bowtie \Delta') \cup \Delta'\}$;
3. $\text{old}_\Delta := \Delta'$; $\Delta := \{< J >\} \times \Delta'$; $\Delta' := \emptyset$;
 $\text{old_All}' := \text{All}'$; $\text{All}' := \{< J >\} \times \text{All}'$; $\text{All}' := \emptyset$;
4. $J := J + 1$; **goto** 1

Notice that steps 2a and 2b have been simplified w.r.t. translation template 1, because Q' is not recursive and then the iteration cycle is useless (indeed it would reach saturation on the first step and then exit on the second one).

5 Optimization of Datalog++ queries

A systematic study of query optimization techniques is realizable on the basis of the concrete implementation of the iterated stable model procedure discussed in the previous section. We now sketch a repertoire of ad hoc optimizations for Datalog++, by exploiting the particular syntactic structure of programs and queries, and the way they use the temporal arguments.

First of all, we observe that the computations of translation template 3 never terminate. An obvious termination condition is to check that the relations computed at two consecutive stages

are empty. To this purpose, the translation template 3 can be modified by inserting the following instruction between step 2 and 3:

if $\forall p$ defined in $Q : P = old_P = \emptyset$ then stop

A more general termination condition is applicable to *deterministic* cliques, under the assumption that the external calls to the predicates of the clique do not specify particular stages, i.e., external calls are of the form $p(-, \dots)$. In this case, the termination condition above can be simplified as follows:

if $\forall p$ defined in $Q : P = old_P$ then stop

Forgetful-fixpoint computations. In many applications (e.g., modeling updates and active rules [22, 10]) queries are issued with reference to the final stage only (which represents the commit state of the database). Such queries often exhibit the form

$p(I, X), \neg p(s(I), -)$

with the intended meaning “find the value X of p in the final state of p ”. This implies that (i) when computing the next stage, we can forget all the preceding states but the last one (see [23]), and (ii) if a stage I such that $p(I, X), \neg p(s(I), -)$ is unique, we can quit the computation process once the above query is satisfied. For instance, the program in Example 7 with the query $\Delta(I, X), \neg \Delta(s(I), -)$ computes the leaf nodes at maximal depth in a breadth-first visit of the graph rooted in a . To the purpose of evaluating this query, it suffices to (i) keep track of the last computed stage only, (ii) exit when the current Δ is empty. The code for the program of Example 7 is then optimized by:

(i) replacing step 3 with:

3. $old_Delta' := Delta'; Delta' := \emptyset;$
 $old_All' := All'; All' := \emptyset;$

i.e., dropping the instructions that record previous stages;

(ii) insert between steps 2 and 3 the instruction:

if $Delta' = \emptyset$ then $\Delta := \{ < J - 1 > \} \times old_Delta'$; stop

Another interesting case occurs when the answer to the query is distributed along the stages, e.g., when we are interested in the answer to a query such as $\Delta(-, X)$, which ignores the stage argument. In this case, we can collect the partial answers via a gathering predicate defined with a copy-rule. For instance, the all predicate in Example 7 collects all the nodes reachable from a . Then the query $all(I, X), \neg all(s(I), -)$, which is amenable for the described optimization, is equivalent to the query $\Delta(-, X)$, which on the contrary does not allow it. Therefore, by (possibly) modifying the program with copy-rules for the all predicate, we can apply systematically the space optimized forgetful-fixpoint.

Delta-fixpoint computations. We already mentioned the presence of a *copy-rule* in Example 7:

$all(s(I), X) \leftarrow all(I, X), \Delta(s(I), -)$.

Its effect is that of copying all the tuples from the stage I to the next one, if any. We can avoid such useless space occupation, by maintaining for each stage only the modifications which are to be applied to the original relation in order to obtain the actual version. For example, the above rule represents no modification at all, and hence it should not have any effect; indeed, it suffices to keep track of the additions to the original database requested by the other rule:

$$\text{all}(I, X) \leftarrow \Delta(I, X).$$

which can be realized by a supplementary relation all^+ containing, at each stage, the new tuples produced. In the case that we replace the *copy-rule* with a *delete-rule* of the form:

$$\text{all}(s(I), X) \leftarrow \text{all}(I, X), \Delta(s(I), -), \neg q(X).$$

we need simply to keep track of the negative contribution due to literal $\neg q(X)$, which can be stored in a relation all^- . Each $\text{all}(I, \dots)$ can then be obtained by integrating $\text{all}(0, \dots)$ with all the $\text{all}^+(J, \dots)$ and $\text{all}^-(J, \dots)$ atoms, with $J \leq I$. This method is particularly effective when $\text{all}(0, \dots)$ is a large relation. To illustrate this point, let us assume that the program of Example 7 is modified by adding a new exit rule for relation all :

$$\text{all}(0, X) \leftarrow r(X).$$

where r is an extensional predicate. The resulting code is then the following:

0. $J := 0$; $\text{old_}\Delta' := \emptyset$; $\Delta' := \emptyset$;
 $\text{old_All}^+ := \emptyset$; $\text{All}^+ := \emptyset$;
1. if $J = 0$ then $\Delta' := \pi_Y(\sigma_{X=a}(G(X, Y)))$;
- 2a. $\Delta' := \Delta' \cup \pi_Y(\text{old_}\Delta'(X) \bowtie G(X, Y) \bowtie r(Y) \cup \text{old_All}^+(Y))$;
- 2b. $\text{All}^+ := \Delta' \cup \text{old_All}^+$;
3. $\text{old_}\Delta' := \Delta'$; $\Delta := \{ \langle J \rangle \} \times \Delta'$; $\Delta' := \emptyset$;
 $\text{old_All}' := \text{All}'$; $\text{All} := \{ \langle J \rangle \} \times \text{All}'$; $\text{All}' := \emptyset$;
4. $J := J + 1$; goto 1

In this way we avoid the construction of relation All , i.e., the replication of relation r at each stage. In fact, All is reconstructed on the fly when needed (step 2a).

Side-effect computations. A direct combination of the previous two techniques gives rise to a form of *side-effect* computation. Let us consider, as an example, the nondeterministic ordering of an array performed by swapping at each step any two elements which violate ordering. Here, the array $a = \langle a_1, \dots, a_n \rangle$ is represented by the relation a with extension $a(1, a_1), \dots, a(n, a_n)$.

$$\begin{array}{ll} \text{ar}(0, P, Y) & \leftarrow a(P, Y). \\ \text{swp}(I, P1, P2) & \leftarrow \text{ar}(I, P1, X), \text{ar}(I, P2, Y), X > Y, P1 < P2, \text{choice}((I), (P1, P2)). \\ \text{ar}(s(I), P, X) & \leftarrow \text{ar}(I, P, X), \neg \text{swp}(I, P, -), \neg \text{swp}(I, -, P), \text{swp}(I, -, -). \\ \text{ar}(s(I), P, X) & \leftarrow \text{ar}(I, P1, X), \text{swp}(I, P1, P). \\ \text{ar}(s(I), P, X) & \leftarrow \text{ar}(I, P1, X), \text{swp}(I, P, P1). \\ ? \text{ar}(I, X, Y), \neg \text{ar}(s(I), -, -) & \end{array}$$

At each stage i we nondeterministically select an unordered pair x, y of elements, delete the array atoms $ar(i, p1, x)$ and $ar(i, p2, y)$ where they appear, and add the new atoms $ar(s(i), p1, y)$ and $ar(s(i), p2, x)$ representing the swapped pair. The query allows a forgetful-fixpoint computation (in particular, stage selected by the query is unique), and the definition of predicate ar is composed by delete-rules and an add-rules. This means that at each step we can (i) forget the previously computed stages (but the last), and (ii) avoid copying most of relation ar , keeping track only of the deletions and additions to be performed. If the requested update are immediately performed, the execution of the proposed program, then, boils down to the efficient iterative computation of the following (nondeterministic) Pascal-like program:

```
while  $\exists I a[I] > a[I + 1]$  do  $swap(a[I], a[I + 1])$  od
```

6 Conclusions

The work reported in this paper, concerning fixpoint/operational semantics and optimization of a logic database language for non deterministic and nonmonotonic reasoning, constitutes the starting point for an actual implemented system. Such project is currently in progress, on the basis of the *LDL++* system developed at UCLA. We plan to incorporate the proposed optimization into the *LDL++* compiler, to the purpose of

- evaluating how *effective* the proposed optimizations are for realistic *LDL++* programs, i.e., whether they yield better performance or not,
- evaluating how *applicable* the proposed optimizations are for realistic *LDL++* programs, i.e., how often they can be applied,
- experimenting the integration of the proposed optimizations with the classical optimization techniques, such as magic-sets.

References

1. K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Proc. Workshop on Found. of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufman, 1988.
2. N. Arni, K. Ong, S. Tsur, and C. Zaniolo. *LDL++: A Second Generation Deductive Databases Systems*. Technical report, MCC Corporation, 1993.
3. A. Brogi, V. S. Subrahmanian, and C. Zaniolo. The Logic of Totally and Partially Ordered Plans: a Deductive Database Approach. *Annals of Mathematics in Artificial Intelligence*, 19:59–96, 1997.
4. J. Chomicki. Temporal deductive databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, pages 294–320. Benjamin Cummings, 1993.
5. M Gelfond and V. Lifchitz. The Stable Model Semantics for logic programming. In *Proc. of the 5th Int. Conf. on Logic Programming*, pages 1070–1080, 1988.
6. F. Giannotti, S. Greco, D. Saccà, and C. Zaniolo. Programming with non Determinism in Deductive Databases. *Annals of Mathematics in Artificial Intelligence*, 19:97–125, 1997.
7. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. *Datalog++: A basis for Active Object-Oriented Databases*. Technical Report C97-011, The CNUCE Institute, 1997.

8. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Query Answering in Non deterministic Non monotonic logic databases. In *Proc. Conference on Flexible Query-Answering Systems FQAS'98*, 1998. To Appear.
9. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Semantics of Non deterministic Non monotonic logic databases. In *Conference on Computer Science Logic CSL'98*, 1998. To be presented.
10. F. Giannotti, G. Manco, and D. Pedreschi. A Deductive Data Model for Representing and Querying Semistructured Data. In *Proc. 5th Int. Conf. on Deductive and Object-Oriented Databases (DOOD97)*, December 1997.
11. F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Non-Determinism in Deductive Databases. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases (DOOD91)*, volume 566 of *Lecture Notes in Computer Science*, pages 129–146, 1991.
12. F. Giannotti, D. Pedreschi, and C. Zaniolo. Semantics and Expressive Power of Non Deterministic Constructs for Deductive Databases. Technical Report C96-04, The CNUCE Institute, 1996. Submitted.
13. D. Kemp, K. Ramamohanarao, and P. Stuckey. ELS programs and the efficient evaluation of non-stratified programs by transformation to ELS. In *Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases (DOOD95)*, pages 91–108, 1995.
14. R. Krishnamurthy and S. Naqvi. Non-deterministic Choice in Datalog. In *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, pages 416–424, 1988.
15. B. Lüdascher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proc. 7th COMAD Int. Conf. on Management of Data*. Tata McGraw-Hill, 1995.
16. M. Nanni. Nondeterminism and XY-Stratification in Deductive Databases (in Italian). Master's thesis, Department of Computer Science University of Pisa, 1997.
17. S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, 1989.
18. T. C. Przymusiński. Every logic program has a natural stratification and an iterated fix point model. In *Proc. 8th ACM PODS Symposium on Principles of Database Systems*, pages 11–21, 1989.
19. K. A. Ross. Modular Stratification and Magic Sets for Datalog Program with Negation. *Journal of ACM*, 41(6):1216–1266, November 1994.
20. D. Saccà and C. Zaniolo. Stable Models and Non-determinism in Logic Programs with Negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.
21. J. D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I-II. Computer Science Press, 1989.
22. C. Zaniolo. Active Database Rules with Transaction Conscious Stable Model Semantics. In *Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases (DOOD95)*, volume 1013 of *Lecture Notes in Computer Science*, pages 55–72, 1995.
23. C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: The *LDL++* Approach. In *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases (DOOD93)*, volume 760 of *Lecture Notes in Computer Science*, 1993.
24. C. Zaniolo, S. Ceri, C. Faloutsos, R.T Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufman, 1997.