Modelling the Future with Event Choice DATALOG

Antonella Guzzo[1] and Domenico Saccà[1,2]

[1] DEIS Dept, University of Calabria, 87036 Rende, Italy
{guzzo,sacca}@deis.unical.it
[2] ICAR-CNR, 87036 Rende, Italy sacca@isi.cs.cnr.it

Abstract. This paper presents a rule-based declarative database language which extends stratified DATALOG to express events and nondeterministic state transitions, by using various types of choice to model uncertainty in dynamic rules. The proposed language, called Event Choice DATALOG (DATALOG$^{ev!}$ for short), is particularly suitable to formulate queries on the evolution of a knowledge base on the basis of a given sequence of events that are envisioned to occur in the future. The semantics of a DATALOG$^{ev!}$ program is given by means of rule rewriting which transforms it into a DATALOG program with choice and a particular form of XY-stratification, called XYZ-stratification, that uses a pair of temporal arguments to handle evolution.

1 Introduction

DATALOG is a rule-based declarative database language that is amenable to very efficient implementation as demonstrated by a number of prototypes of deductive database systems [10, 11, 20, 28]. Many proposals have been issued to extend DATALOG in order to support nonmonotonic queries, mainly by means of various forms of negation in the bodies of the rules. The first solution was *stratified negation* [4, 9, 33], which has a simple, intuitive semantics leading to efficient implementation. Unfortunately, this type of negation has a reduced expressive power for it can only express a proper subset of fixpoint queries.

The next step toward greater expressive power was to remove the condition that there is no recursion through negation. In this framework, a dramatic leap in expressive power is provided by the concept of *stable model* [13] but this gain is not without complications. Indeed the usage of unrestricted negation in programs is often neither simple nor intuitive, and, for example, might lead to writing programs that have no total stable models.

As argued in [18], advances in model-theoretic semantics for nonmonotonic DATALOG should be achieved without surrendering the naturalness and efficiency of stratified negation. A solution is a language where the usage of stable model semantics is disciplined to refrain from abstruse forms of unstratified negation which may lead to undefinedness or unnecessary computational complexity. The core of a desirable database language should be stratified DATALOG, that is extended with only predefined types of non-stratified negation, hardwired into ad-hoc constructs.

A first construct for capturing a controlled form of unstratified negation was the *choice*, whose semantics was defined in terms of stable models in [30] by exploiting the nondeterminism implicit in the notion of such models. The combination of choice with extrema aggregates is investigated in [19] and many other facets of logic programming with choice are detailed in [15]. Recently, the problem of extending choice DATALOG to capture various complexity classes of database boolean queries and to express search and optimization queries has been addressed respectively in [18] and in [17].

Another interesting direction in extending the expressive power of DATALOG by a disciplined usage of unstratified negation is represented by XY-stratification which was first introduced in [37] and has later been used to model updated and active rules [35, 36]. The recursive predicates of an XY-stratified program have a temporal argument which is used to enforce local stratification, a weak form of stratification introduced in [27].

In this paper we present a new language, called Event Choice DATALOG (DATALOG$^{\mathrm{ev!}}$ for short), which combines the capability of choice to express nondeterminism with the power of XY-stratification to model evolution — actually we introduce an extension of XY-stratification, called XYZ-stratification, which uses a second temporal argument to keep track of sub-units of time for a finer control of the evolution. Indeed, when an event occurs in a DATALOG$^{\mathrm{ev!}}$ program, a number of actions may be triggered which are almost instantaneous and must be completed before the occurrence of the next event — the second time argument is used to keep track of the execution of such actions between events.

DATALOG$^{\mathrm{ev!}}$ is particularly suitable to express events and dynamic rules for modelling the evolution of a initial knowledge base, expressed with the declarative style of stratified DATALOG, and to make queries on possible future states of the knowledge base as results of events that are envisioned to happen.

Many other languages for modelling events and transactions exist, e.g., transaction logic [5, 6], event calculus [23, 22], dynamic logic programming [2], LUPS [3], LOCO with events [29]. The novelty of our proposal is the attempt to model the evolution of knowledge states, triggered by events, using nondeterministic transition rules, convinced as we are that a deterministic approach is not appropriate in modelling the future. Probably it is better to draw various alternative trajectories rather than insisting in enforcing one, which often remains arbitrary, no matter the quantity of optimization is spent to justify it.

DATALOG$^{\mathrm{ev!}}$ supports reasoning about events and actions as it happens for logic-based languages for planning (see [14, 31] for surveys on this topic which has recently received a renewed great deal of interest). Indeed our language has partly been influenced by recent declarative planning languages such as $\mathcal{K}$ [12] and $\mathcal{C}$ action language [16]. We however stress that, at least at this stage, our language does not support planning, i.e., finding a sequence of actions which lead from an initial state to some success state satisfying a given goal. DATALOG$^{\mathrm{ev!}}$ rather enables a different form of reasoning on events, *prediction*, for which the sequence of future events is given together with the goal: since the same event may actually occur in different forms because of the nondeterministic nature of transition rules, the problem consists in verifying whether there exists a particular happening which eventually satisfies the given goal and in returning a possible future state satisfying the goal.

The paper is organized as follows. In Section 2 we present our extensions of choice and of XY-stratification and in Section 3 we illustrate the syntax and the semantics of $\texttt{DATALOG}^{\texttt{ev!}}$, which is given by rewriting the rules to eventually produce an XYZ-stratified program with choice. In Section 4 we introduce queries, discuss their complexity and include meaningful examples. Finally we draw the conclusion and discuss further lines of research in Section 5.

2 Extended Choice and XYZ-Stratification

We assume that the reader is familiar with basic notions of logic programming and `DATALOG` [1, 24, 32].

A *logic program* P is a finite set of rules r of the form $H(r) \leftarrow B(r)$, where $H(r)$ is an atom (*head* of the rule) and $B(r)$ is a conjunction of literals (*body* of the rule). A rule with empty body is called a *fact*. The *ground instantiation* of P is denoted by $\mathit{ground}(P)$; the *Herbrand universe* and the *Herbrand base* of P are denoted by U_P and B_P, respectively.

Let an interpretation $I \subseteq B_P$ be given — with a little abuse of notation we sometimes see I as a set of facts. Given a predicate symbol r in P_D, $I(r)$ denotes the relation $\{t : r(t) \in I\}$. Moreover, $\mathit{pos}(P, I)$ denotes the positive logic program that is obtained from $\mathit{ground}(P)$ by (i) removing all rules r such that there exists a negative literal $\neg A$ in $B(r)$ and A is in I, and (ii) by removing all negative literals from the remaining rules. Finally, I is a (*total*) *stable model* [13] if $I = \mathbf{T}^{\infty}_{\mathit{pos}(P,I)}(\emptyset)$, that is the least fixpoint of the classical *immediate consequence transformation* for the positive program $\mathit{pos}(P, I)$.

Given a logic program P and two predicate symbols p and q, we write $p \rightarrow q$ if there exists a rule where q occurs in the head and there is a predicate in the body, say s, such that either $p = s$ or $p \rightarrow s$. P is *stratified* if for each p and q, if $q \rightarrow p$ holds in it then p does not occur negated in the body of any rule whose head predicate symbol is q, i.e. there is no recursion through negation. Stratified programs have a unique stable model which coincides with the *stratified model*, obtained by partitioning the program into an ordered number of suitable subprograms ('strata') and computing the fixpoints of every stratum in their order [4].

A $\texttt{DATALOG}^{\neg}$ program is a logic program with negation in the rule bodies but without functions symbols. Predicate symbols can be either extensional (i.e., defined by the facts of a database — *EDB predicate symbols*) or intensional (i.e., defined by the rules of the program — *IDB predicate symbols*). A $\texttt{DATALOG}^{\neg}$ program P has associated a relational database scheme $\mathcal{DB}_P$, which consists of all EDB predicate symbols of P. Given a database D on $\mathcal{DB}_P$, the tuples of D are seen as facts added to P; so P on D yields the following logic program $P_D = P \cup \{q(t). : q \in \mathcal{DB}_P \wedge t \in D(q)\}$. The class of all $\texttt{DATALOG}^{\neg}$ programs is simply called $\texttt{DATALOG}^{\neg}$; the subclass of all stratified programs is called $\texttt{DATALOG}^{\neg_s}$.

The complexity of computing a stable model of P_D is measured according to the *data complexity* approach of [8, 34] for which the program is assumed to be constant while the database is variable. It is well known that computing a stable model of a $\mathtt{DATALOG}^{\neg_s}$ program P on a database D can be done in time polynomial on the size of D whereas it requires exponential time (unless $\mathcal{P} = \mathcal{NP}$) in case P is not stratified. Actually, in the latter case, deciding whether there exists a stable model or not is $\mathcal{NP}$-complete[25].

A disciplined form of unstratified negation is the *choice* construct, which is used to enforce functional dependency (FDs) constraints on rules of a logic program and to introduce a form of nondeterminism. The formal semantics of the choice can be given in terms of stable model semantics as shown next [30].

Let a *choice rule* r with a choice construct[3] be given:

$$r : A \leftarrow B(Z),\ \mathit{choice}((X),(Y)).$$

where, $B(Z)$ denotes the conjunction of all the literals in the body of r that are not choice constructs, Z is the list of all variables occurring in B, and X, Y denote lists of variables such that $X \cap Y = \emptyset$ and $X, Y \subseteq Z$ — note that X can be empty and in this case, it is denoted by "`( )`". The construct $\mathit{choice}((X),(Y))$ prescribes that the set of all consequences derived from r, say R, must respect the FD $X \to Y$, thus if two consequences happen to have the same values for X but different ones for Y then only one consequence, nondeterministically selected, will be eventually derived.

The formal semantics of choice is given in terms of stable models by replacing the above choice rule with the following rules:

$$\begin{array}{l} A \leftarrow B(Z),\ \mathit{chosen}_r(W). \\ \mathit{chosen}_r(W) \leftarrow B(Z),\ \neg \mathit{diffChoice}_r(W). \\ \mathit{diffChoice}_r(W) \leftarrow \mathit{chosen}_r(W'),\ Y \neq Y'. \end{array}$$

where $W = X \cup Y$, W' is the list of variables obtained from W by replacing each $V \notin X$ with a new variable V (e.g. by priming those variables), and $Y \neq Y'$ is true if $V \neq V'$, for some variable $V \in Y$ and its primed counterpart $V' \in Y'$.

Next we introduce a simple variation of choice, denoted by $\mathit{choiceAny}()$, which nondeterministically selects one consequence. Thus this construct is a shorthand of $\mathit{choice}((),(Z))$, where Z is the list of all variables occurring in the rule body, according to the meaning of the FD $\emptyset \to Z$.

Two powerful variations of the choice are described in [19]: $\mathit{choiceLeast}((X),C)$ and $\mathit{choiceMost}((X),C)$, where C is a single variable defined on an ordered domain, which select respectively the minimal and the maximal value for C, while enforcing the FD $X \to C$. A rule with $\mathit{choiceLeast}$, say

$$r : a(Y) \leftarrow B(Z),\ \mathit{choiceLeast}((X),C).$$

[3] In general a choice rule may contain more than one choice construct in the body but for this paper one will be enough.

is rewritten as

$$a'(Y,X,C) \leftarrow B(Z).$$
$$nonMin(X,C) \leftarrow a'(Y,X,C), a'(Y',X,C'), C' < C.$$
$$a(Y) \leftarrow a'(Y,X,C), \neg nonMin(X,C).$$

A straightforward but useful variation of *choiceLeast* is $choiceMin(C)$ which selects the consequences with the minimal value for C and is a shorthand of $choiceLeast((W),C)$, where W is the list of all variables in the body except C. In a similar way we define *choiceMost* and *choiceMax*.

We point out that *choiceLeast*, *choiceMost* and their variations are indeed deterministic.

A $\texttt{DATALOG}^{\neg}$ program P with the above choice constructs is called an *extended choice program.* We say that P is *stratified modulo choice* if, by considering choice atoms as extensional atoms, the program results stratified. We denote with $sv(P)$ (*standard version* of P) the program obtained by rewriting the choice constructs as above. Given a database D, if P is stratified modulo choice and there is no recursion through the predicates *choiceLeast*, *choiceMost*, *choiceMin* and *choiceMax*, then the stable models of $sv(P)_D$ are in general multiple but the existence of at least one as well as its computation in polynomial time is guaranteed.

Another approach in disciplining unstratified negation is to add a distinguished temporal argument to recursive predicate symbols and to allow only two types of recursive rules:

1. *X-rule* when the temporal argument of the head predicate is the same variable as in all temporal arguments of the recursive literals in the body which only occur positive;
2. *Y-rule* when the head temporal argument is $T+1$ and all temporal arguments of the literals in the body are equal to T, where T is a variable — in this case negation is allowed.

This extension, introduced in [37], is called *XY-Stratification* and has been used to model updated and active rules [35, 36]. XY-stratified programs are indeed locally stratified [27] and, therefore, there exists a unique stable model although it can be infinite because of the temporal argument. Nevertheless, for practical applications it is possible to include restrictions into a XY-stratified program in order to guarantee both the finiteness of the stable model and its computation in polynomial time.

We now propose a simple extension of XY-stratification that will turn to be useful for implementing the language we shall present in the next section. We add a second temporal argument which keeps track of sub-units of time, thus the temporal dimension is handled by a pair of temporal arguments (T_1, T_2) whose ordering is lexicographic — thus $(T_1, T_2) \leq (T'_1, T'_2)$ if (i) $T_1 < T'_1$ or (ii) $T_1 = T'_1$ and $T_2 \leq T'_2$.

A program is XYZ-stratified[4] if there exists three types of recursive rules:

[4] The "Z" cames from both the presence of a third rule and the initial of Carlo, the father of XY-stratification.

1. *X-rule* if all temporal argument pairs in the rule is the same pair of variables;
2. *Y-rule* when the head temporal argument pair is $(T_1, T_2{+}1)$ and all temporal argument pairs of the literals in the body are equal to (T_1, T_2);
3. *Z-rule* when the head temporal argument pair is $(T_1 + 1, 0)$ and every temporal argument pair of the literals in the body is equal to (T_1, T), where T is any variable or constant.

It is easy to see that also XYZ-stratified are locally stratified and have a unique stable model. Also in this case this model can be infinite and its fixpoint computation could even be transfinite.

In the next section we shall present a language whose semantics is based on XYZ-stratification and on extended choices for which both existence and finiteness of stable models is guaranteed and one of them can be computed in polynomial time. We note that the combination of XY-stratification and choice has been first used in [7] to model various planning problems.

3 $\texttt{DATALOG}^{\mathsf{ev!}}$: Event Choice DATALOG

In this section we present *Event Choice* `DATALOG`, called $\texttt{DATALOG}^{\mathsf{ev!}}$. In addition to classical EDB and IDB `DATALOG` predicate symbols, the language includes:

– *dynamic* (*DDB*) predicate symbols which are used for facts to be asserted upon the occurrences of events;
– *event* predicate symbols having an additional argument which provides the time dimension — an event predicate atom has the format $p(X)@(T)$, where X is a list of arguments and T is the time argument stating that the event $p(X)@(T)$ occurs at the time T with the properties described in X.

A $\texttt{DATALOG}^{\mathsf{ev!}}$ program comprises: (i) the *static knowledge* that is a $\texttt{DATALOG}^{\neg_s}$ program with IDB predicate symbols in the rule heads and DB (i.e., EDB, IDB and DDB) predicates symbols in the bodies, and (ii) a number of *event definitions.* An event definition consists of the *event declaration* within brackets and of one or more *transition rules.* It has the following format:

$$[e(X)@(T)] \quad t_1 \quad \cdots \ t_k$$

where $e(X)@(T)$ is the event which enables the transition rules t_i $(1 \leq i \leq k$, $k > 0)$ as soon as it occurs. A transition rule is of the form:

$$E_1\& \cdots \& E_n\& \, A_1\& \cdots \& A_m \leftarrow B \otimes C_1 \otimes \cdots \otimes C_s.$$

where

1. E_i $(1 \leq i \leq n,\ n \geq 0)$ is an event atom that is triggered if the body of the transition rule is true; E_i has the format $g(X)$`++`, where g is an event symbol — informally, the event $g(X)$ will occur at the time $T + \delta$ where δ is a sub-unit of time that is used to enable micro transitions for a finer

tuning of the program evolution[5]; we stress that sub-units of time cannot be directly handled but only incremented using the above syntax;
2. A_i ($1 \le i \le m$, $m > 0$) is a DDB atom that is made true when the rule body is satisfied — we require that at least one DDB atom must be present in the transition rule head;
3. B is a conjunction of DB literals (negation is allowed in the body of the transition rule);
4. C_i ($1 \le i \le s$, $s \ge 0$) is an extended choice atom (i.e., *choice*, *choiceAny*, *choiceLeast*, *choiceMin*, *choiceMost*, *choiceMax*); the choices are performed in the order they appears in the rule (i.e., as it often happens during an evolution, ordering is indeed relevant).

To simplify the notation, we use some syntactic sugar for writing negative literals in the rule bodies: $\neg a(X)$, stands for $\neg a'(Y)$, where a' is defined by the new rule:

$$a'(Y) \leftarrow a(X).$$

where Y is the list of all non-anonymous variables occurring in X.

Example 1. **Subgraphs.** We are given two EDB predicate symbols `arc` and `node`, defined by a number of suitable facts, which encode an undirected graph, say G, with weights on the arcs. There are two event predicate symbols: `startSub` and `contSub`, which allow to construct subgraphs of G which are indeed rooted trees. Each subgraph is numbered, say with an integer k; this number, the root node, say r, and each subgraph arc, say (i, j), are stored in DDB atoms as `subGraph(k)`, `root(k, r)` and `subArc(k, i, j)`, respectively.

The static knowledge defines a number of properties for the subgraphs which will be constructed by means of ad-hoc events:

– given a subgraph S_G numbered with L, `reached(L, X, C)` states that there is a path from the root of S_G to the node X in G consisting only of arcs in S_G and the total sum of their weights is C; for every reached node X, `leaf(L, X)` is true if no arc in S_G exits from it:

$$\begin{array}{ll}
\texttt{reached(L,X,0)} & \leftarrow \texttt{root(L,X)}.\\
\texttt{reached(L,X,C)} & \leftarrow \texttt{reached(L,Y,C}'\texttt{)},\texttt{subArc(L,Y,X)},\texttt{arc(Y,X,C}''\texttt{)},\texttt{C} = \texttt{C}' + \texttt{C}''.\\
\texttt{leaf(L,X)} & \leftarrow \texttt{reached(L,X,\_)},\neg\texttt{subArc(L,X,\_)}.
\end{array}$$

– if some node in G is not reached from the root of S_G then `noAllReached(L)` is true; if some reached node X has two incoming (resp., outgoing) arcs in S_G then `twoIn(L)` (resp., `twoOut(L)`) is true:

$$\begin{array}{ll}
\texttt{noAllReached(L)} & \leftarrow \texttt{subGraph(L)},\texttt{node(X)},\neg\texttt{reached(L,X,\_)}.\\
\texttt{twoIn(L)} & \leftarrow \texttt{subArc(L,Y,X)},\texttt{subArc(L,Y}'\texttt{,X)},\texttt{Y} \neq \texttt{Y}'.\\
\texttt{twoOut(L)} & \leftarrow \texttt{subArc(L,X,Y)},\texttt{subArc(L,X,Y}'\texttt{)},\texttt{Y} \neq \texttt{Y}'.
\end{array}$$

[5] Note that, in contrast with the C++ jargon, $g(X)$++ triggered at the time T is not equal to $g(X)@(T + 1)$

– S_G is declared a spanning tree (i.e., $\mathtt{spanTree(L)}$ is true) if all nodes in G are reached from its root and no two of its arcs enter into the same node; in addition, if no two of its arcs exit from the same node, then S_G is declared a Hamiltonian path (i.e., $\mathtt{hPath(L)}$ is true); furthermore, if the Hamiltonian path can be extended to form a circuit in the graph G, $\mathtt{hCircuit(L,C)}$ is true and C is the sum of all weights in the circuit:

$$\begin{array}{ll}
\mathtt{spanTree(L)} & \leftarrow \mathtt{subGraph(L)}, \neg\mathtt{noAllReached(L)}, \neg\mathtt{twoIn(L)}. \\
\mathtt{hPath(L)} & \leftarrow \mathtt{spanTree(L)}, \neg\mathtt{twoOut(L)}. \\
\mathtt{hCircuit(L,C)} & \leftarrow \mathtt{hPath(L)}, \mathtt{root(L,X)}, \mathtt{leaf(L,Y)}, \mathtt{reached(L,Y,C')}, \\
 & \mathtt{arc(Y,X,C'')}, \mathtt{C} = \mathtt{C'} + \mathtt{C''}.
\end{array}$$

– a subgraph S'_G is declared distinct from S_G if there exists at least one arc which is not shared by them:

$$\begin{array}{l}
\mathtt{distinct(L,L')} \leftarrow \mathtt{subArc(L,X,Y)}, \neg\mathtt{subArc(L',X,Y)}. \\
\mathtt{distinct(L,L')} \leftarrow \mathtt{subArc(L',X,Y)}, \neg\mathtt{subArc(L,X,Y)}.
\end{array}$$

The dynamic part of the program consists of two events:

– the first event $\mathtt{startSub}$ begins the construction of a subgraph numbered by L and of type K, provided that no other subgraph with the same number has been already constructed; any node in the graph is chosen as root and the event *contSub* is triggered in order to select the arcs to be included in the subgraph:

$$\begin{array}{l}
[\mathtt{startSub(L,K)@(T)}] \\
\quad \mathtt{contSub(L,K)}\texttt{++}\ \& \\
\quad \mathtt{subGraph(L)}\ \&\ \mathtt{root(L,X)} \leftarrow \neg\mathtt{subGraph(L)},\ \mathtt{node(X)}\ \otimes\ \mathtt{choiceAny()}.
\end{array}$$

– the event $\mathtt{contSub}$ continues the construction of the subgraph according to the properties associated to the type K:
  1. $K = st$: then the subgraph will be constructed with arcs which are not included into previous subgraphs and will be a spanning tree if the graph remains connected also without such arcs;
  2. $K = mst$: then, if the graph is connected, the subgraph will be a minimum spanning tree — arcs included into previous subgraphs can be reused in this case;
  3. $K = hp$: then the subgraph will be a path, hopefully Hamiltonian — also in this case there is no restriction on which arcs can be used;

$$\begin{array}{ll}
\multicolumn{2}{l}{[\mathtt{contSub(L,K)@(T)}]} \\
\quad \mathtt{contSub(L,K)}\texttt{++}\ \&\ \mathtt{subArc(L,X,Y)} & \leftarrow \mathtt{K} = \mathtt{st},\ \mathtt{reached(L,X,\_)}, \\
 & \mathtt{arc(X,Y,\_)},\ \neg\mathtt{subArc(\_,X,Y)}, \\
 & \neg\mathtt{reached(L,Y,\_)}\ \otimes\ \mathtt{choice((Y),(X))}. \\
\quad \mathtt{contSub(L,K)}\texttt{++}\ \&\ \mathtt{subArc(L,X,Y)} & \leftarrow \mathtt{K} = \mathtt{mst},\ \mathtt{reached(L,X,\_)}, \\
 & \mathtt{arc(X,Y,C)},\ \neg\mathtt{reached(L,Y,\_)} \\
 & \otimes\ \mathtt{choiceMin(C)}\ \otimes\ \mathtt{choiceAny()}. \\
\quad \mathtt{contSub(L,K)}\texttt{++}\ \&\ \mathtt{subArc(L,X,Y)} & \leftarrow \mathtt{K} = \mathtt{hp},\ \mathtt{reached(L,X,\_)},\ \mathtt{arc(X,Y,\_)}, \\
 & \neg\mathtt{reached(L,Y,\_)},\ \neg\mathtt{subArc(L,X,\_)} \\
 & \otimes\ \mathtt{choiceAny()}.
\end{array}$$

□

Given a $\texttt{DATALOG}^{\texttt{ev!}}$ program P and a database D, the *standard version* $sv(P_D)$ of P_D is the choice XYZ-stratified program obtained from P_D by applying the following rewriting:

1. Replace each rule in the static definition (including EDB facts), say
$$p(X) \leftarrow q_1(Y_1), \ldots, q_n(Y_n).$$
where $n \geq 0$, with the rule
$$p'(X, T, T1) \leftarrow q_1'(Y_1, T, T1), \ldots, q_n'(Y_n, T, T1), \mathit{time}(T, T1).$$
where *time* is an ad-hoc predicate symbol which keeps track of all the times when events occur, so that the static rule may continue to hold in the actual future — observe that the second argument of *time* corresponds to the subunit of time;
2. for each DDB predicate symbol p, say with arity n, add the following rules:
$$\begin{array}{l} p'(X, T, T1+1) \leftarrow p(X, T, T1), \mathit{time}(T, T1+1). \\ p'(X, T+1, 0) \leftarrow p(X, T, T1), \mathit{time}(T', 0), T' > T. \end{array}$$
where X is a list of n distinct variables, thus dynamic atoms that are asserted at a certain time will hold in the future by inertia — note that, to avoid infinite models, we only consider times of the future that will actually occur;
3. add the fact $\mathit{time}(0, 0)$. in order to enable the initial state of the program;
4. for each event definition, say with declaration $[e(Z)@(T)]$, and for each transition rule, say:
$$\begin{array}{ll} g_1(X_1)\texttt{++} \,\&\, \cdots \,\&\, g_k(X_k)\texttt{++} \,\&\, & \\ p_1(W_1) \,\&\, \cdots \,\&\, p_n(W_n) & \leftarrow B(Y) \otimes C_1 \otimes \cdots \otimes C_m. \end{array}$$
where Y is the list of all variables occurring in the conjunction B, we proceed as follows:
   (a) we rewrite $B(Y)$ as $B'(Y, T, T1)$ by adding the pair of time arguments to each literal in B;
   (b) for each i, $1 \leq i \leq m$, we replace C_i with C_i' in the following way: if C_i equals $\mathit{choice}((U), (V))$, $\mathit{choiceLeast}((U), V)$ or $\mathit{choiceMost}((U), V)$, we simply replace (U) with $(U, T, T1)$; furthermore, $\mathit{choiceAny}()$ is rewritten as $\mathit{choice}((T, T1), (Y))$; finally $\mathit{choiceMin}(V)$ and $\mathit{choiceMax}(V)$, are rewritten respectively as $\mathit{choiceLeast}((Y', T, T1), V)$ and $\mathit{choiceMost}((Y', T, T1), V)$, where $Y' = Y - \{V\}$;
   (c) we add the following $m + 1$ rules:
$$\begin{array}{l} b_0(Y, T, T1) \leftarrow B'(Y, T, T1), e(Z, T, T1). \\ b_i(Y, T, T1) \leftarrow b_{i-1}(Y, T, T1),\ C_i'. \ \ (1 \leq i \leq m) \end{array}$$
   which enforce ordering of the m choices by means of the a suitable stratification;

(d) we include the following n rules:

$$p_i'(W_i, T, T1+1) \leftarrow b_m(Y, T, T1). \quad (1 \leq i \leq n)$$

for registering the consequences of the dynamic rule at one sub-unit of time after the occurrence of the event — thus there is a "small" delay from the occurrence of the event e to the derivation of possible consequences;

(e) we use the following n rules to keep track of all times when the consequences of the event have been occurred — note that a time of happening an event is registered only if at least one consequence has been derived:

$$time(T, T1+1) \leftarrow p_i'(W_i, T, T1+1), \neg p_i'(W_i, T, T1) \quad (1 \leq i \leq n)$$

(f) the triggers will be possibly activated using the following rules:

$$g_i(X_i, T, T1+1) \leftarrow b_m(Y, T, T1), time(T, T1+1) \quad (1 \leq i \leq k)$$

— note that a trigger cannot run forever because of the constraint that it can be activated only if at least a consequence of the dynamic rule has been derived.

An (*evolution*) E on a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program P is a list of ground events, ordered by the temporal argument, i.e., E is a sequence of events that are envisioned to happen. Let $sv(E)$ denote the set of facts $e(x, t, 0).$, one for each event $e(x)@(t)$ in E.

Example 2. **Subgraph evolution.** A possible evolution for the program in Example 1 is:

$$[\mathtt{startSub}(1, \mathtt{hp})@(0), \mathtt{startSub}(2, \mathtt{st})@(1), \mathtt{startSub}(3, \mathtt{mst})@(2),$$
$$\mathtt{startSub}(4, \mathtt{hp})@(2)]$$

The construction of a path will be initiated at the time 0 and will be completed before the construction of a spanning tree at the time 1 — observe that the spanning tree will not use the arcs of the path. At the time 2, after completing the spanning tree (or subtree if the is no way to cover the graph using the available arcs), two subgraphs are constructed in parallel: a minimal spanning tree of the graph and another path which is not necessarily different from the first one as, in this case, we may reuse the arcs included in previous subgraphs. We pinpoint that most likely the two paths will not be Hamiltonian although we aim at getting such a property. □

Theorem 1. *Let P be a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program and E an evolution for it. Then for each database D on $\mathcal{DB}_P$, $sv(P_D) \cup sv(E)$ admits at least one stable model, every stable model is finite and computing a stable model of $sv(P_D)$ can be done in time polynomial on the size of D.* □

Let us now introduce some notation that will used in the next section, where we shall show how to query a $\texttt{DATALOG}^{\texttt{ev!}}$ program by predicting its possible evolutions on the basis of a list of envisioned future events. Given a stable model M of $sv(P_D) \cup sv(E)$, for each time t, $max_{subT}(M, t)$ is a partial function which returns the maximal value t' such that $time(t, t')$ is in M. Moreover, given a time t, M_t denotes the subset of M whose atoms have temporal arguments $(t1, t2)$ for which $(t1, t2) < (t + 1, 0)$.

4 $\texttt{DATALOG}^{\texttt{ev!}}$ Queries

A query on a $\texttt{DATALOG}^{\texttt{ev!}}$ program P is of the form $\langle E, G, R\rangle$ where

- E is an evolution — say that t^E_{max} is the last time of the events in E;
- G (*goal*) is a list of query conditions of the form $[g_1@(t_1), \ldots, g_n@(t_n)]$ $(n \geq 0)$, where $0 \leq t_1 < t_2 < \ldots < t_n \leq t^E_{max}$; each g_i can be:
  - $\exists(A)$, where A is a conjunction of ground DB literals;
  - $\forall(A)$, where A is a (possibly negated) conjunction of ground DB literals;
  - *opt*$(X : A)$, where *opt* $= min$ or max, X is a variable and A is a conjunction of EDB, IDB and DDB literals containing no variables except X;
- R (*result*) is a list $[r_1(X_1)@(t_1), \ldots, r_m(X_m)@(t_m)]$, $m > 0$ and $t_1 \leq \ldots \leq t_m \leq t^E_{max}$, where r_i is any IDB or DDB predicate symbol, say with arity k_i, and X_m is a list of k_i terms.

Given a query $Q = \langle H, G, R\rangle$ on a $\texttt{DATALOG}^{\texttt{ev!}}$ program P and a database D on $\mathcal{DB}_P$, a stable model M of $sv(P_D) \cup sv(E)$ is *Q-filtered* if for each $g_i@(t_i)$ in the goal G:

1. if $g_i = \exists(A)$, then $A'(t_i, max_{subT}(M, t_i))$ must be true in M, where A' is obtained from the conjunction A by adding the temporal ground arguments $(t_i, max_{subT}(M, t_i))$ to each literal in it;
2. if $g_i = \forall(A)$, then $A'(t_i, max_{subT}(N, t_i))$ must be true in all stable models N of $sv(P_D) \cup sv(E)$ such that $M_{t_i} = N_{t_i}$ — informally, A must be true at the time $(t_i, max_{subT}(t_i))$ for all possible choices which can be made after the time $(t_{i-1} + 1, 0)$ or the time (0,0) if $i = 1$, even though only one of such choices will be retained;
3. if $g_i = min(X : A)$, then (i) $A'(t_i, max_{subT}(M, t_i))$ is true in M for some value of X and (ii) for each stable model N of $sv(P_D) \cup sv(E)$ for which $M_{t_i} = N_{t_i}$ and $A'(t_i, max_{subT}(N, t_i))$ is true in N for some value of X, $min_X(M) \leq min_X(N)$, where $min_X(M)$ is the minimal value which makes true A' in M and $min_X(N)$ is defined accordingly — informally, all choices from $(t_{i-1} + 1, 0)$ on will be tried in order to select the ones which minimizes X at the time $(t_i, max_{subT}(t_i))$;
4. the case $g_i = max(X : A)$ is defined as in the previous point by replacing min_X with max_X.

Given a query $Q = \langle E, G, R\rangle$ on a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program P, where $R = [r_1@(t_1), \ldots, r_m@(t_m)]$, and a database D on $\mathcal{DB}_P$, an (nondeterministic) *answer* of Q on D, denoted by $Q(D)$, is either:

1. the list of relations $[\mathbf{r}(X_1)_1, \ldots, \mathbf{r}(X_i)_m]$ such that $\mathbf{r}_i = \{x_i | r'_i(x_i, t_i, max_{subT}$ $(M, t_i)) \in M$ and x_i unifies with $X_i\}$, where r' is the temporal version of the predicate symbol r and M is a *Q-filtered* stable model of $sv(P_D) \cup sv(E)$,
2. or (ii) the empty list if there is no Q-filtered stable model.

We are now ready to present the results about the complexity of $\mathtt{DATALOG}^{\mathtt{ev!}}$ queries. The reader can refer to [21, 26] for excellent sources of information on basic concepts of complexity. We point out that a query Q is a multivalued function whose domain is the family of all possible databases. Given a database D, the complexity of computing an answer of $Q(D)$ is measured with respect to the size of D.

Theorem 2. *Let $Q = \langle E, G, R\rangle$ be a query on a $\mathtt{DATALOG}^{\mathtt{ev!}}$ program P. Then for each database D on $\mathcal{DB}_P$,*

1. *computing an answer of $Q(D)$ is in $\mathcal{FP}^{\mathcal{NP}}$, i.e., it is computable in polynomial time on the size of D by a deterministic Turing machine that can query an oracle in $\mathcal{NP}$ a polynomial number of times;*
2. *if G is the empty list, then computing an answer of $Q(D)$ is in $\mathcal{FP}$, i.e., it can be done in polynomial time on the size of D.* □

Example 3. **Subgraph Queries.** Consider the program in Example 1. The query $\langle\, [\mathtt{startSub(1, st)@(0)}], [\,], [\mathtt{subArc(1, X, Y)@(0)}]\,\rangle$ on a connected graph asks to return a spanning tree. Obviously a solution is found in time polynomial on the size of the graph.

Another polynomial query is $\langle\, [\mathtt{startSub(1, mst)@(0), startSub(2, st)@(1)}]$, $[\,]$, $\mathtt{subArc(1, X, Y)@(0), subArc(2, X, Y)@(1)}]\,\rangle$ which asks to return first a minimum spanning tree and then, if the remaining arcs keep the graph connected, a spanning tree. By replacing the goal $[\,]$ with $[\exists\neg\mathtt{noAllReached(2)@(1)}]$, the query will ask to select the minimum spanning tree which does not disconnect the graph, if one exists — note that the computation may now have an exponential cost.

Next we present a rather elaborated query which cannot be answered in polynomial time unless $\mathcal{P} = \mathcal{NP}$. We want to find an optimal travelling salesman path (TSP), provided that there is no other TSP with the same cost (i.e., there exactly one optimal TSP). The query

$$\begin{array}{l}\langle\ [\mathtt{startHP(1)@(0),\ startHP(2)@(1)}],\ [\mathtt{min(CT : hCircuit(1, CT))@(0)},\\ \quad \forall\neg(\mathtt{hCircuit(2, CT)} \wedge \mathtt{hCircuit(1, CT)} \wedge \mathtt{distinct(1, 2)})\mathtt{@(1)}],\\ \quad [\mathtt{subArc(1, X, Y, C)@(0)}]\ \rangle\end{array}$$

returns the optimal TSP provided that it is unique. □

The next example shows that $\mathtt{DATALOG}^{\mathtt{ev!}}$ queries can be effectively used to implements greedy algorithms. To simplify the writing we shall use stratified aggregates whose semantics can be easily included into our formal ground. We also introduce an additional construct for disciplining unstratified negation, denoted by $\mathit{select}((X),(D))$, where D is a conjunction of literals, which selects for each group of consequences with the same values for X, those which satisfy D if any or, otherwise, all of them[6]. Given a rule with *select*, say:

$$r : a(Y) \leftarrow B(Z),\ \mathit{select}((X),(D)).$$

we perform the following rewriting:

$$\begin{array}{l} \mathit{satD}(X) \leftarrow B(Z),\ D. \\ a(Y) \leftarrow B(Z),\ D. \\ a(Y) \leftarrow B(Z),\ \neg \mathit{satD}(X). \end{array}$$

We point out that the construct $\mathit{select}((X),(D))$ into a transition rule must first be rewritten as $\mathit{select}((X,T,T1),(D))$.

A simple variation of *select* is $\mathit{possibly}(D)$ which is a shorthand of $\mathit{select}((Y),(D))$, where Y is the list of all variables in the body. In the case of a transition rule $\mathit{possibly}(F)$ is rewritten as $\mathit{select}((Y,T,T1),(D))$.

Obviously both *select* and *possibly* perform deterministic selections.

Example 4. **Team building.**

We are given projects, employees and skills represented by the following EDB facts:

```
project(P#, Budget, Duration, NofSkills, TeamSize).
employee(E#, Skill#, Salary, Sex).
skill(Skill#, Description).
```

For a given project, we want to set up a team of employees such that:

1. all skills, except at most two, be present in the group — this property is modelled by the following static rule:

   $\mathtt{skilled(P)} \leftarrow \mathtt{project(P,\_,\_,Nskill,\_)}, \mathtt{nSkills(P,N)}, \mathtt{N} \geq \mathtt{Nskill} - 2.$

   where *nSkills* is an IDB predicate symbol which counts the number of skills present in the project team;
2. the sum of the salaries of the employees working in the same team may not exceed the budget and must be minimum — this condition is expressed by the static rule:

   $$\begin{array}{ll} \mathtt{budgeted(P,TSal)} \leftarrow & \mathtt{project(P,Budget,\_,\_,\_)},\ \mathtt{TSal} = \\ & \mathtt{sum}\{\mathtt{Sal} : \mathtt{employee(E,\_,Sal,\_)} \wedge \mathtt{inTeam(P,E)}\}, \\ & \mathtt{TSal} \leq \mathtt{Budget}. \end{array}$$

   where *inTeam* is a DDB predicate symbol which stores the employees enrolled in a project;

[6] Note that "all" is not a typo here - if the condition D is not satisfied we want to accept all consequences rather than reject them

3. at least the 50% of the employees in the group be women — the corresponding static rule is:

$$\begin{aligned}\texttt{pairOpportunity(P)} \leftarrow\ & \texttt{project}(\texttt{P}, \_, \_, \_, \texttt{Nteam}),\ \texttt{N} = \\ & \texttt{count}\{\texttt{E} : \texttt{inTeam}(\texttt{P}, \texttt{E}) \wedge \texttt{employee}(\texttt{E}, \_, \_, \texttt{female})\}, \\ & \texttt{N} \geq \texttt{Nteam} * 0, 5.\end{aligned}$$

The static knowledge is completed by:

1. the rule for counting the number of skills available in a project team:

$$\begin{aligned}\texttt{nSkills}(\texttt{P}, \texttt{N}) \leftarrow\ & \texttt{project}(\texttt{P}, \_, \_, \_, \_),\ \texttt{N} = \\ & \texttt{count}\{\texttt{Skill} : \texttt{employee}(\texttt{E}, \texttt{Skill}, \_, \_) \wedge \texttt{inTeam}(\texttt{P}, \texttt{E})\}.\end{aligned}$$

2. a rule for checking whether the project is staffed or not:

$$\texttt{staffed}(\texttt{P}) \leftarrow \texttt{project}(\texttt{P}, \_, \_, \_, \texttt{Nteam}),\ \texttt{Nteam} = \texttt{count}\{\texttt{E} : \texttt{inTeam}(\texttt{P}, \texttt{E})\}\}.$$

3. a rule stating that an employee E, enrolled at the time TP in a project with duration D, is still engaged with that project at the time T if $T \leq TP + D$:

$$\begin{aligned}\texttt{engaged}(\texttt{E}, \texttt{T}) \leftarrow\ & \texttt{inTeam}(\texttt{P}, \texttt{E}), \texttt{projectStart}(\texttt{P}, \texttt{TP}), \texttt{project}(\texttt{P}, \_, \texttt{D}, \_, \_), \\ & \texttt{T} \leq \texttt{TP} + \texttt{D}.\end{aligned}$$

The dynamic definition is:

$$\begin{aligned}&[\texttt{startProject}(\texttt{P})@(\texttt{T})] \\ &\quad \texttt{staffing}(\texttt{P})\texttt{++}\ \&\ \texttt{projectStart}(\texttt{P}, \texttt{T}) \leftarrow \neg\texttt{projectStart}(\texttt{P}, \_).\end{aligned}$$

$$\begin{aligned}&[\texttt{staffing}(\texttt{P})@(\texttt{T})] \\ &\quad \texttt{staffing}(\texttt{P})\texttt{++}\ \&\ \texttt{inTeam}(\texttt{P}, \texttt{E}) \quad \leftarrow \texttt{employee}(\texttt{E}, \_, \_, \_), \neg\texttt{engaged}(\texttt{E}, \texttt{TP}), \\ &\qquad\qquad \texttt{projectStart}(\texttt{P}, \texttt{TP}), \neg\texttt{staffed}(\texttt{P}), \\ &\qquad\qquad \otimes\ \texttt{choiceAny}().\end{aligned}$$

The trigger `staffing` includes any employee into the project team provided that it is not already engaged with another project and the team is not yet fully staffed. The query

$$\begin{aligned}&\langle\, [\texttt{startProject}(\texttt{p1})@(1), \texttt{startProject}(\texttt{p2})@(4)], \\ &\ [\texttt{min}(\texttt{TSal} : \texttt{skilled}(\texttt{p1}) \wedge \texttt{bugdeted}(\texttt{p1}, \texttt{TSal}) \wedge \texttt{pairOpportunity}(\texttt{p1}))@(1), \\ &\ \ \texttt{min}(\texttt{TSal} : \texttt{skilled}(\texttt{p2}) \wedge \texttt{bugdeted}(\texttt{p2}, \texttt{TSal}) \wedge \texttt{pairOpportunity}(\texttt{p2}))@(4)] \\ &\ [\texttt{inTeam}(\texttt{p1}, \texttt{E})@(1), \texttt{inTeam}(\texttt{p2}, \texttt{E})@(4)]\,\rangle\end{aligned}$$

returns the team compositions for the project $p1$ and $p2$ which satisfy the above conditions on budget, skills and pair opportunity and minimize the total costs of team member salaries as well.

As computing an answer could be expensive or even unfeasible, it may be convenient to release some constraints using a greedy approach. A possible solution is shown next.

First of all we add a static rule checking whether the skill of the employee E is already present in the team project:

$$\begin{aligned}\texttt{existsSameSkill}(\texttt{P}, \texttt{E}) \leftarrow\ & \texttt{employee}(\texttt{E}, \texttt{Skill}, \_, \_), \texttt{inTeam}(\texttt{P}, \texttt{E1}), \\ & \texttt{employee}(\texttt{E1}, \texttt{Skill}, \_, \_).\end{aligned}$$

We replace the definition of the event staffing with:

$$\begin{array}{ll} \multicolumn{2}{l}{[\mathtt{staffing(P)@(T)}]} \\ \quad\mathtt{staffing(P)\text{++}}\ \& & \\ \quad\mathtt{inTeam(P,E)} & \leftarrow \mathtt{employee(E,\_,Sal,Sex)}, \neg\mathtt{staffed(P)}, \\ & \quad \mathtt{projectStart(P,TP)}, \neg\mathtt{engaged(E,TP)} \\ & \quad \otimes\,\mathtt{possibly}(\neg\mathtt{skilled(P)} \wedge \neg\mathtt{existsSameSkill(P,E)}) \\ & \quad \otimes\,\mathtt{possibly}(\mathtt{Sex = female} \wedge \neg\mathtt{pairOpportunity(P)}) \\ & \quad \otimes\,\mathtt{choiceMin(Sal)} \otimes \mathtt{choiceAny()}. \end{array}$$

The order of selection constructs correspond to the relevance we assign to the various properties: first the availability of skills, then the pair opportunity and finally a minimal cost. The query

$$\langle\,[\mathtt{startProject(p1)@(1)}, \mathtt{startProject(p2)@(4)}], [\,], [\mathtt{inTeam(p1,E)@(1)}, \mathtt{inTeam(p2,E)@(4)}]\,\rangle$$

returns an approximate solution of the optimal staffing problem. □

5 Conclusion

In this paper we have presented an extension of DATALOG with events and choice, called $\mathtt{DATALOG}^{\mathtt{ev!}}$, which is particularly suitable to express queries on the evolution of a knowledge base on the basis of a given sequence of events that are envisioned to occur in the future. The language allows to model a number of alternative potential evolutions of a program by means of dynamic rules which assert both dynamic facts and actions (i.e., triggered events), using the capability of choice to express nondeterminism. We stress that $\mathtt{DATALOG}^{\mathtt{ev!}}$ does not support planning (i.e., finding a sequence of actions which lead from an initial state to some success state satisfying a given goal) but a different form of reasoning on events, that we call *prediction*: given a sequence of events and a goal, since an event may actually happen in different forms because of the nondeterministic nature of dynamic rules, the problem consists in verifying whether there exists an evolution which eventually satisfies the given goal and in returning one of the future states satisfying the goal.

The semantics of $\mathtt{DATALOG}^{\mathtt{ev!}}$ is given by means of rewriting rules which produce a DATALOG program with choice and a new form of stratification, called XYZ-stratification, which add to classical XY-stratification a second temporal argument to keep track of sub-units of time for a finer control of the evolution.

We conclude by mentioning some lines of further research:

1. allowing negation in the head of dynamic rules to enlarge the expressive power of $\mathtt{DATALOG}^{\mathtt{ev!}}$ even though termination will not anymore guaranteed;
2. introducing disjunction in the head of dynamics rules, in particular for triggers — this extension will enable $\mathtt{DATALOG}^{\mathtt{ev!}}$ to deal with planning as well;
3. including new forms of choice which make alternative selections using data mining techniques applied to previous histories — thus the past will be used to model the future;
4. specializing the language to support workflows.

References

1. Abiteboul, S., Hull, R., and Vianu, V., *Foundations of Databases.* Addison-Wesley. 1994.
2. Alferes, J.J., Leite, J.A., Pereira, L.M., and Przymusinski T., Dynamic Logic programming. In A. Cohn and L. Schubert, Eds, *KR'98.* Morgan Kaufmann, 1998.
3. Alferes, J.J., Pereira, L.M., Przymusinska, H., and Przymusinski, T.C., LUPS A language for updating logic programs. *Artificial Intelligence*, Vol.138, n1-2), pp. 87–116, 2002.
4. Apt, K., Blair, H., and Walker, A., Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kauffman, Los Altos, USA, 1988, 89-142.
5. Bonner, A.J.,and Kifer, M., An overview of transaction logic, *Theoretical Computer Science*, Vol.113, n2, pp. 205–265, 1994.
6. Bonner, A.J., and Kifer, M., Results on Reasoning about Updates in Transaction Logic, *Lecture Notes in Computer Science*, Vol. 1472, 1998
7. Brogi, A., Subrahmanian, V. S., and Zaniolo, C., Modeling Sequential and Parallel Plans. *Journal of Artificial Intelligence and Mathematics*, Vol.19, n3, April 1997.
8. Chandra, A., and Harel, D., Structure and Complexity of Relational Queries. In *Journal of Computer and System Science*, Vol. 25, n1, pp. 99–128, 1982.
9. Chandra, A., and Harel, D., Horn Clauses Queries and Generalizations. In *Journal of Logic Programming*, Vol. 25, n1, pp. 1–15, 1985.
10. Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., and Zaniolo, C., The LDL System Prototype. In *IEEE TKDE*, Vol. 2, n1, pp. 76–90, 1990.
11. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., The KR system dlv: Progress report, comparison and benchmarks.In Proc. Sixth Int. Conf. on Priciples of Knowledge Representation and Reasoning (KR98),A.G.Cohn, L.Schubert, and S.Shapiro, Eds, Morgan Kaufmann Publishers, 406-417.
12. Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A., A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transaction on Computational Logic*, 2002, to appear.
13. Gelfond, M., and Lifschitz, V., The Stable Model Semantics for Logic Programming. *Proc. 5th Int. Conf. on Logic Programming*, pp.1070-1080, 1988.
14. Gelfond, M., and Lifschitz, V., Action languages. *Electronic Transactions on Artificial intelligence*, Vol. 2, n 3-4, pp.193-210, 1998.
15. Giannotti, F., Pedreschi, D., and Zaniolo, C., Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases. *Journal of Computer and System Sciences, 62*, 1, pp. 15–42, 2001.
16. Giunchiglia, E., and Lifschitz, V., An Action Language Based on Causal Explanation: Preliminary Report. In *Proc. of the Fifteenth Nat. Conf. on Artificial Intelligence (AAAI'98)*, pp. 623–630, 1998.
17. Greco, S., and Saccà, Search and Optimization Algorithm in Datalog. In *Computational Logic: From Logic Programming into the Future*, Kakas A., Sadri F. (edrs), Springer Verlag, 2002.
18. Greco, S., Saccà, D., and Zaniolo, C., Extending Stratified Datalog to Capture Complexity Classes Ranging from P to QH. In *Acta Informatica*, Vol. 37 N10, pp 699-725, July 2001.
19. Greco, S., and Zaniolo, C., Greedy Algorithms in Datalog. in *Proc. Int. Joint Conf. and Symp. on Logic Programming*, 1998, pp. 294–309.

20. Ilkka Niemelä, Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Journal of Artificial Intelligence and Mathematics*, Vol.25, N3-4, 1999.
21. Johnson, D. S., A Catalog of Complexity Classes. In *Handbook of Theoretical Computer Science*, Vol. 1, J. van Leewen (ed.), North-Holland, 1990.
22. Kowalski, R.A., Database updates in event calculus. *Journal of Logic Programming*, Vol.12, pp.121-146, January 1992.
23. Kowalski, R., A., and Sergot, M.J., A Logic-Based Calculus of Events, *New Generation Computing*, Vol 4, pp. 267, 1986.
24. Lloyd, J., *Foundations of Logic Programming.* Springer-Verlag, 1987.
25. Marek, W., Truszczynski, M., Autoepistemic Logic. *Journal of the ACM*, Vol. 38, No. 3, pp. 588–619, 1991.
26. Papadimitriou, C. H., *Computational Complexity.* Addison-Wesley, Reading, MA, USA, 1994.
27. Przymusinski T.C., On the Declarative and procedural Semantics of Stratified Deductive Databases, in *Foundations of Deductive Databases and Logic Programming*, (J.W. Minker, ed.), pp. 193-216, Morgan Kaufman, USA, 1988.
28. Ramakrisnhan, R., Srivastava, D., and Sudanshan, S., CORAL — Control, Relations and Logic. In *Proc. of 18th Conf. on Very Large Data Bases*, pp. 238-250, 1992.
29. Saccà, D., Verdonk, B., and Vermeir, D., Evolution of Knowledge Bases. In *Proc. EDBT*, 1992, pp. 230-244.
30. Saccà, D., and Zaniolo, C., Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. ACM Symp. on Principles of Database Systems*, 1990, pp. 205-218.
31. Turner, H., A logic of universal causation, *Artificial Intelligence*, Vol.113, n1-2, pp.87–123, 1999.
32. Ullman, J. K., *Principles of Data and Knowledge-Base Systems*, Vol.1-2. Computer Science Press, New York, 1988.
33. Van Gelder, A., Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, Vol. 6, n. 1, pp. 109–133, 1989.
34. Vardi, M., The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pp. 137–146, 1982.
35. Zaniolo, C., Transaction-Conscious Stable Model Semantics for Active Database Rules. In *Proc. Int. Conf. on Deductive Object-Oriented Databases*, 1995.
36. Zaniolo, C., Active Database Rules with Transaction-Conscious Stable Model Semantics. In *Proc. of the Conf. on Deductive Object-Oriented Databases*, pp.55–72, LNCS 1013, Singapore, December 1995.
37. Zaniolo, C., Arni, N., and Ong, K., Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach, *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases*, 1993.