Event Choice Datalog: A Logic Programming Language for Reasoning in Multiple Dimensions

Gianluigi Greco
DEIS
Università della Calabria
87030 Rende - Italy
ggreco@deis.unical.it

Antonella Guzzo
DEIS
Università della Calabria
87030 Rende - Italy
guzzo@deis.unical.it

Domenico Saccà
ICAR-CNR and DEIS, Univ. della Calabria
87030 Rende - Italy
sacca@icar.cnr.it

Francesco Scarcello
DEIS
Università della Calabria
87030 Rende - Italy
scarcello@deis.unical.it

ABSTRACT
This paper presents a rule-based declarative database language which extends DATALOG to express events and nondeterministic state transitions, by using the choice construct to model uncertainty in dynamic rules. The proposed language, called Event Choice Datalog (DATALOG$^{\text{ext}}$ for short), provides a powerful mechanism to formulate queries on the evolution of a knowledge base, given a sequence of events envisioned to occur in the future. A distinguished feature of this language is the use of multiple spatio-temporal dimensions in order to model a finer control of evolution. A comprehensive study of the computational complexity of answering DATALOG$^{\text{ext}}$ queries is reported.

Categories and Subject Descriptors
I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving—Logic Programming; D.3.2 [Programming Languages]: Language Classifications—Constraint and logic languages

General Terms
Languages

Keywords
Logic Programming, Knowledge Representation

1. INTRODUCTION

Finding a suitable declarative framework for modeling and reasoning about actions is a problem that has received a great deal of interest in the past years. Indeed, logic-based languages (see, e.g., [20, 43]) developed in the context of logics for knowledge representation might be profitably exploited for defining and solving planning problems, that often arise in AI applications. Traditional declarative approaches for planning fall into three distinct categories: situation calculus ([33]), temporal reasoning (see, e.g., [36]) and event calculus ([26]). Several recent proposals exploit, instead, logic programming and, specifically, the answer set programming paradigm for developing domain-independent planning languages [30]. Besides the very declarative modeling features of logic programming, the most interesting aspect is that, since answer sets represent the solution of the planning problem, planners may be easily implemented with the support of efficient answer set engines such as GnT [24], DLV [28], Smodels [41], DeReS [12], XSB [39], and ASSAT [31]. The language $K$ [16] is a prototypical representative of the languages exploiting such an approach. In fact, it is completely based on principles and methods of logic programming and its main feature is the ability of dealing with incomplete knowledge, i.e., of modeling scenarios when the designer has a partial knowledge of the world, only.

One of the major limitations of the language $K$, as well as of most of logic-based languages for reasoning about actions, is the lack of an explicitly support for time, in the planning process. Moreover, the few logic-languages dealing with timed actions rarely consider the possibility of dealing with multiple time or spatial units. Typically, they are suitable extensions of the event calculus (EC) [9].

Multiple time units are also called time granularities, in the temporal database community (see, e.g., [8]). The basic idea is to distinguish a number of time dimensions at different scales, in order to model the validity of properties over coarser or finer time intervals. Note that the simple solution of mapping all dimensions into the finest scale does not always work, as the coarser dimensions may impose restrictions on time validity, and because it can lead to a tremendous blowup in the size of the problem instance. In summary, multiple time units can be profitably used for two main purposes:

- For modeling in a more natural way a large class of problems, such as those involving planning tasks. In fact, it has been recognized that the ability to reason with multiple time granularities is an important feature, as all the human activities are essentially related to multiple units, such as weeks, days, and hours. (See [8], for an overview of different proposals and the definition of a unifying model for multiple time granularities.)

- For dealing with plans at different levels of details. Each time dimension may be seen as a conceptual dimension, and we can employ a main dimension for reasoning about complex activities, and some auxiliary time dimension for modeling the execution of possible subtasks these activities are made of. Note that, since these subtasks are executed in a different dimension, they can be seen as instantaneous, as far as the main time scale is concerned. This can be particularly useful if, at the main activities level, we are only interested in the effects of subtasks, rather than on their temporal properties, as shown in a subsequent example.
Note that the latter issue is also a viable way for planning in Hierarchical Task Networks (HTNs) [38]. The HTN framework is an approach to planning where problem-specific knowledge is used to remedy the computational intractability of classical planning. This knowledge comes in the form of task decomposition directives: the planner is given a set of methods telling how a high-level task can be decomposed into lower-level tasks. First attempts to define languages able to explicitly deal with time and HTN planning problems have been done in [19], where suitable extension of the Golog/ConGolog [29] languages (based on situation calculus) are designed.

In this paper we tackle the above knowledge representation issues in the logic programming context, and we propose a new language, called Event Choice \textsc{Datalog} (\textsc{Datalog} for short), for modeling and querying action theories with different time granularities. The language can be used for defining declaratively sub-tasks that may eventually be combined in order to solve complex activities, in a way that is completely transparent to the user. We point out that combining programs (or program fragments) is not natural in logic programming, as the union of two programs may have unexpected semantics. Conversely, an interesting peculiarity of \textsc{Datalog} is its \textit{modularity}, which makes it useful for many applications, e.g., for the HTN planning. Indeed, note that any program performing some planning task can be reused in more complex systems, by making it working in a proper subunit of time, without any interference with previously defined modules.

We will show that \textsc{Datalog} is well suited for modeling and reasoning about complex dynamic systems in real applicative scenarios, and can be useful for simulation and design purposes.

1.1 Overview of the Language

In a nutshell, \textsc{Datalog} is a language for modeling the evolution of knowledge states, triggered by events and guided by nondeterministic transition rules. Its main features are:

- **Event Activation Rules:** The language models transitions among states of the world by exploiting the notion of event, in the same spirit of \textsc{C} [22]. The occurrence of an event enables the application of a rule that may modify the state by asserting or retracting some facts (fluents), and may trigger other events to occur in the future. The language also supports the interaction with external events. This latter feature is particularly useful for simulating and reasoning about possible scenarios, as we shall describe in our motivating examples.

- **Choice constructs:** The ability to deal with the nondeterminism has been recognized as a key feature of logic-based languages. However, an undisciplined use of unstratified negation and/or disjunction leads to higher computational complexities and to hard-to-read programs. For this reason, \textsc{Datalog} programs are stratified, but their rules may contain choice atoms, that provide nondeterministic features. In particular, if we are not interested in a particular outcome (temporal evolution) of the program, the choice construct is able to model a don’t-care form of nondeterminism.

Thus, \textsc{Datalog} combines the capability of the choice construct to express nondeterminism (possibly, don’t-care nondeterminism), with the event activation rules, used for modeling events occurring at certain specified time instants. Moreover, one can make queries on possible future states of the knowledge base, given some list of events that are envisioned to happen. In the rest of this section, we give a brief overview of the language by considering an example of complex planning problem. We present this example in two steps, starting from the classical \textsc{Blocksworld} problem [40], and then exploiting its encoding as a subtask of a more involved planning problem. In a subsequent section, we will also show another application in a different domain.

1.1.1 Planning with Events and Choice

We have a table and a set of blocks. The table can hold arbitrarily many blocks, while each block can hold at most one other block. Initially, blocks \texttt{a} and \texttt{b} are on the table, while block \texttt{c} is on the top of \texttt{a}. We can move a block at a time to the table or on the top of another block, provided that its top is empty. We want to find a sequence of moves leading to the configuration in which \texttt{a} is on the table, \texttt{b} is on the top of \texttt{a}, and \texttt{c} on the top of \texttt{b} — see Figure 1.

The first component of a \textsc{Datalog} program used for modeling such program is the \textit{background knowledge} expressed as a set of facts, denoted by \texttt{EDB} (extensional database), which are assumed to do not change over the time. These facts specifies the objects involved in the modelled domain. In our example, \texttt{EDB} consists of the facts

\begin{verbatim}
block(a). block(b). block(c).
\end{verbatim}

The second component is a set of fluents, denoted by \texttt{DDB} (dynamic database). These facts can be dynamically asserted or retracted during the time, on the basis of the occurrence of events. In the example, we can assume to have dynamic facts of the form \texttt{on(X,Y)}, which specifies that block \texttt{X} is on the top of \texttt{Y}. E.g., the initial scenario shown in the leftmost part of Figure 1 is represented by the following \texttt{DDB}

\begin{verbatim}
on(a,table). on(b,table). on(c,a).
\end{verbatim}

The third component is a set of dynamic rules, denoted by \texttt{D-KB} (dynamic knowledge base). These rules are essentially aatalog rules (possibly with stratification) whose predicates are equipped with a time argument. Rules in \texttt{D-KB} are used for expressing properties that depend on the time, and hence they may relate status of the world at different time units. For instance, a rule of the form

\begin{verbatim}
move()@<S,3>
\end{verbatim}
p(X)@\(T \leftarrow q(X)\}@\((T + 2)\) imposes that predicate \(p(X)\) is true two instants of time after predicate \(q(X)\) is. In the special case that all predicates in a dynamic rule deal with the same time instant, such a rule can be used for representing static knowledge, i.e., invariant over the time — in these cases, the time argument is often stripped off. For instance, in our running example, D-KB contains the rules

\[
\begin{align*}
\text{fixed}(B) & \leftarrow \text{on}(B', B), \text{block}(B), \\
\text{goodLocation}(D) & \leftarrow \text{block}(D), \neg \text{fixed}(D), \\
\text{goodLocation}(D) & \leftarrow D = \text{table}, \\
\text{done}() & \leftarrow \text{on}(a, \text{table}), \text{on}(b, A), \text{on}(c, A).
\end{align*}
\]

Intuitively, \(\text{fixed}(B)\) is false at any given time \(T\) if the block \(B\) has no other block on the top of it and, hence, it can be freely moved; \(\text{goodLocation}(D)\) is true if either \(D\) is the table or a block without blocks on it; \(\text{done}()\) is true if the desired final condition has been reached.

The most important component of our language consists in the specification of the event activation rules. These rules state that whenever a given event is (internally or externally) triggered, a set of actions will be performed. In our example, we only consider an event requiring the move of a block.

\[
\begin{align*}
\text{move}()@\&T & \leftarrow \neg \text{on}(S, X), \neg \text{on}(S, D) \leftarrow \neg \text{done}(). \\
\text{on}(S, X), \neg \text{fixed}(S), \\
\text{goodLocation}(D), D \neq S, \\
\text{choiceAny}().
\end{align*}
\]

Intuitively, when \(\text{move}()\) is triggered at time \(T\), we check for the condition in the body of the rule. Notice that the predicates in this body do not have an explicit time argument; we will use such a shorthand in the case the time arguments are the same as the triggering time of the event, i.e., if we are looking at the state of the world at the time the event occurs. Specifically, in the above rule, we check whether the planning has not been yet completed (\(\text{done}()\) is false), and whether there is a block \(S\) that can be moved on the top of a good location. Obviously, there are several possible choices, i.e., several blocks can be moved to several locations. Then, the predicate \(\text{choiceAny}()\) is a directive of our language ensuring that only one of these possibilities is non-deterministically chosen. After the choice is done, the status of the world is updated according to the head of the rule, that is, \(\text{on}(S, X)\) is retracted and \(\text{on}(S, D)\) is asserted for representing the move of \(S\). Moreover, since we have not yet completed the planning, the event \(\text{move}()\) is internally triggered again. If we assume that such moves require ten time units, e.g., ten seconds, then we may trigger this event at time \((T + 10)\).

The interesting feature of the above formalization is that the user writes simple rules involving non-deterministic actions. The focus in writing DATALOG^{ex} programs goes only in properly defining one step of transition only. Then, the non-deterministic transitions ensure that all the possible evolutions can be considered. Note that in this example we are interested in those particular sequences of moves leading to the achievement of the goal. This can be easily specified by means of the last component of DATALOG^{ex} i.e., its query language. In order to query a program, we have to specify a set of (external) events that are envisioned to happen. In our case, this list consists of the single event \(H = \{\text{move}()@0\}\), specifying that the planning starts at time \(0\). Then, a query of the form \(Q = \exists^\exists T \text{done}()\) will be true iff there exists a possible way for achieving the desired final condition within time \(T\). As we can see from Figure 1, this query will be evaluated true for any \(T \geq 30\) (seconds). We next see how the blocksworld program can be reused for solving a subgoal of a more complex planning problem.

### Figure 2: Labyrinth.

#### 1.1.2 Multidimensional Planning

Let us consider a complex scenario, where we have to control a robot that enters a labyrinth (the grid in Figure 2) and should find the exit within a given time bound. In our example, the robot is in the position \((0, 3)\) and must arrive in position \((4, 0)\). As far as this “main” problem is concerned, we are only interested in counting the number of steps in the robot’s escaping path, so that we care only at the spatial dimensions — for instance, we may want to minimize the length of the path. Therefore, in this case, the main dimension is the space or, equivalently, the number of steps, rather than the time.

To make this example more realistic, we consider the case where some walls may be removed by the robot, in order to pass across them. However, this removal is not trivial. Rather, it involves the solution of a blocksworld problem, modeling the feasible ways of breaking the wall. For instance, in Figure 2, we allow the robot to pass across the wall in position \((6, 5)\), if it is able to solve an instance of the blocksworld problem associated with this position in at most 30 seconds. Thus, as far as these subtasks are concerned, we are interested in actual time dimensions.

It is worthwhile noting that this scenario is prototypical of all those situations where we have to realize some goals involving the achievement of further (sub-)goals. Our encoding of this problem shows how DATALOG^{ex} allows us to reuse in a simple way the program that models the blocksworld problem as a module of the full robot program. Moreover, note that we deal here with two dimensions: the length of the path and the execution time of the subtasks.

The static knowledge consists of facts of the form \(\text{wall}(X, Y)\), asserting the presence of a wall in the position of coordinates \((X, Y)\).

The dynamic database DDB comprises an atom \(\text{pos}(X, Y)\) that defines the current position of the robot in the labyrinth (initially, we assert \(\text{pos}(0, 3)\)), and atoms of the form \(\text{moveableWall}(X, Y)\), asserting the presence of a movable wall, which can be traversed by solving an instance of the blocksworld problem. (E.g., in Figure 1, \(\text{moveableWall}(6, 5)\) is in DDB.)

Moreover, D-KB consists of the rules

\[
\begin{align*}
\text{arrived()} & \leftarrow \text{pos}(4, 0). \\
\text{walk}(n, X, Y + 1) & \leftarrow \text{pos}(X, Y), Y < 6, \neg \text{wall}(X, Y + 1). \\
\text{walk}(n, X, Y - 1) & \leftarrow \text{pos}(X, Y), Y > 0, \neg \text{wall}(X, Y - 1). \\
\text{walk}(n, X - 1, Y) & \leftarrow \text{pos}(X, Y), X > 0, \neg \text{wall}(X - 1, Y). \\
\text{walk}(n, X + 1, Y) & \leftarrow \text{pos}(X, Y), X < 6, \neg \text{wall}(X + 1, Y).
\end{align*}
\]

The first rule is used for determining whether the tasks of the robot has been accomplished. Predicate \(\text{walk}\) contains instead the next location of the robot after a walk — notice, that we check whether
exploit the evolution of the system (in this case, of the robot state). Then, we desired evolutions, and hence the moves for reaching the goal. In the syntax of DATALOG are reported in Section 2. We introduce the multidimensional and its model theoretic semantics, by introducing the notions of temporal and stationary model. In Section 6, we formulate queries on DATALOG programs and analyze their complexities. Finally, in Section 7, we present related work, discuss the main novelties of our language, and draw our conclusions.

2. PRELIMINARIES

A Datalog program $P$ is a finite set of rules 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 $ground(P)$; 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 $P$, $I(r)$ denotes the relation $\{ t : r(t) \in I \}$. Moreover, $pos(P, I)$ denotes the positive logic program that is obtained from $ground(P)$ 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 [20] if $I = T^\infty_{pos(P, I)}(\emptyset)$, i.e., it is the least fixpoint of the classical immediate consequence transformation for the positive program $pos(P, I)$. The set of all the stable models of a given program $P$ is denoted by SM($P$).

Given a 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, then $\neg p$ does not occur in the body of any rule whose head predicate symbol is $q$, i.e., there is no recursion through negation. The class of all DATALOG programs is simply called DATALOG; the subclass of all stratified programs is called DATALOG$^{st}$.

Note that stratified programs have a unique stable model that can be computed in polynomial time. However, they allow us to express only deterministic queries. If we need the ability to deal with nondeterminism, we have to use programs with unstratified negation. Unfortunately, in this case, the complexity is higher and sometimes programs become hard to read.

A solution to such drawbacks of negation is disciplining its use, by adding to the basic stratified language some special construct that provides nondeterministic features. In this paper, we consider the choice construct [37], that allows us to express choices in logic programs, by enforcing functional dependency (FD) constraints on the consequences of rules.

Let a choice rule $r$ with a choice construct — in general a choice rule may contain more than one choice construct in the body but for this paper one will be enough — be given:

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

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 $X \subseteq Z$ — note that $X$ can be empty and in this case, it is denoted by “()”. The construct $choice((X), (Y))$ prescribes that the set of all consequences derived from $r$, say $R$, must respect the FD $X \rightarrow Y$. Thus, if two consequences have the same values for $X$ but different ones for $Y$ then only one consequence, nondeterministically selected, will be eventually derived.

We denote by choiceAny() the construct $choice((X), (Y))$ that nondeterministically selects one consequence, where $Z$ is the list of all variables occurring in the rule body, according to the meaning of the FD $\emptyset \rightarrow Z$.

A DATALOG program $P$ with choice rules is called an extended choice program. We say that $P$ is stratified modulo choice (or simply stratified) if, by considering choice atoms as extensional atoms, the program results stratified.
3. MULTIDIMENSIONAL DOMAINS

In this paper we consider a multidimensional model of time, that allows us to consider different level of details, often called granularities in the literature – see [23] and [8].

Each time instant is a tuple \( \{t_1, \ldots, t_n\} \), where \( n \) is the current dimension of this instant and each \( t_i \) is a natural number. The time \( 0 \) is a distinguished element standing for the beginning of the time, and is denoted by 0. A (multidimensional) time domain \( T \) is a set of time instants.

Note that this notion of time is very general, as we have just conceptual dimensions, which can model any desired level of details in reasoning about events. For instance, we can employ a main dimension for reasoning about some complex activities, and one auxiliary time dimension for modeling the execution of the various subtasks such activities are made of. Note that these subtasks take in fact some time to be executed, but they can be seen as instantaneous, as far as the main time scale is concerned. This can be particularly useful if, at the complex activities level, we are only interested in the effects of subtasks, rather than on their detailed temporal succession.

We often impose some restriction on the set of time instants, either on the number of dimensions, or on the range of each dimension. For instance, the usual notion of time (in every-day life) is modelled as a multidimensional time domain, where we have a main infinite dimension (encoding, e.g., the number of years after Christ) and a number of bounded range sub-dimensions (encoding, e.g., days, hours, minutes, and seconds). We denote by \( T^{t, m} \) this temporal domain. Moreover, we denote by \( T^{h, w} \) a time domain having the number of dimensions bounded by some number \( m \) and each dimension counts at most \( n \) time instants (range of the dimension), for some number \( n > 0 \); \( T^{h, w} \) has infinite temporal ranges over a finite set of dimensions; \( T^{h, w} \) has infinite set of dimensions with bounded ranges.

All time domains \( T \) are linearly ordered according to the usual lexicographical precedence relationship, that we denote by \( \prec \). We also equip time domains with temporal functions, for incrementing or decrementing the current time of a given amount of time units.

**Definition 3.1** Let \( T \) be a time domain. Then, to each time instant \( t = \{t_1, \ldots, t_n\} \), we can apply one of the following operators, also called temporal functions over \( T \):

- \( t + k \), with \( k \geq 0 \) natural number, that increments (if possible) the time in the current dimension \( n \) of \( k \) units, i.e., it outputs \( \{t_1, \ldots, t_n + k\} \); if the increment \( k \) is not possible, because of some bound on the current dimension \( n \), then \( t + k \) is undefined.

- \( t - k \), with \( k \geq 0 \) natural number, that decrements (if possible) the time in the current dimension \( k \) units, i.e., it outputs \( \{t_1, \ldots, t_n - k\} \); if \( t_n < k \), then \( t - k \) is undefined.

- \( \sup(t) \), which is defined if the current dimension \( n \) is greater than 1, and returns the time instant \( \sup(t) = \{t_1, \ldots, t_{n-1} + 1\} \), i.e., it projects \( t \) onto the preceding dimension \( n - 1 \) and increments the time in that dimension.

- \( \inf(t) \), that outputs \( \{t_1, \ldots, t_{n-1}, t_n, 0\} \), i.e., it creates (if possible) a new time dimension; if \( n \) is the maximum allowed number of dimensions in \( T \), \( \inf(t) \) is undefined.

- \( t \), i.e., the identity function.

Moreover, \( t++ \) and \( t-- \) are shorthand for \( t + 1 \) and \( t - 1 \), respectively.

4. EVENT CHOICE DATALOG

In this section we present Event Choice DATALOG (short: DATALOG\({}^{ev}\)), an extension of DATALOG that is able to deal with events and dynamic knowledge, in a temporal framework with multiple dimensions.

4.1 Syntax

Roughly speaking, all DATALOG\({}^{ev}\) predicates are enriched with an additional argument that provides the time dimension: for any literal \( p \) and each time instant \( t \), \( p @ t \) is true if \( p \) holds at time \( t \). We assume that three sets of constants, variables, and time variables symbols, \( \sigma^{const}, \sigma^{vars}, \) and \( \sigma^{time, vars} \) are given, where the constants symbols are disjoint from the (time) variables symbols. Moreover, let \( T \) be a time domain.

A term \( s \) is an element in \( \sigma^{const} \cup \sigma^{vars} \). Moreover, let \( \sigma^{EDB}, \sigma^{EDB}, \sigma^{IDB}, \) and \( \sigma^{DDB} \) be disjoint sets of predicate symbols, with associated arity (\( \geq 0 \)). Then, an EDB atom has a "classical" format \( p(s_1, \ldots, s_n) \) where \( p \) is a symbol in \( \sigma^{EDB} \) and \( s_1, \ldots, s_n \) are terms. Instead \( DDB \) (dynamic extensional predicates), IDB (intentional predicates), and \( EV \) (event predicates) atoms are of the form \( p(s_1, \ldots, s_n)@f(t) \), where \( p \) is a symbol in \( \sigma^{EDB}, \sigma^{IDB}, \sigma^{DDB} \), respectively, \( n \) is the arity of \( p \), \( s_1, \ldots, s_n \) are terms, \( f \) is a temporal function over the domain \( T \), and \( t \) is a time instant or a time variable in \( \sigma^{time, vars} \).

An EDB, DDB, IDB, or EV literal is either an atom or its negation. The set of all the EDB literals (resp. DDB, IDB, EV), is denoted by \( L_{EDB} \) (resp. \( L_{DDB}, L_{IDB}, L_{EV} \)). Furthermore, for any set of literals \( L \), \( L^+ \) and \( L^- \) denote the sets of its positive and of its negative literals, respectively.

**Definition 4.1** A dynamic rule has the form \( p(X_1, \ldots, X_n)@T ← B_1, \ldots, B_m, \) where \( p(X_1, \ldots, X_n)@T \in L_{EDB}^+, m \geq 0, \) and \( B_1, \ldots, B_m \in L_{EDB} \cup L_{DDB} \cup L_{IDB} \). An event activation rule has the following format:

\[ [\sigma(X_1, \ldots, X_n)@T] \quad TR_1 \ldots TR_k \]

where \( e(X_1, \ldots, X_n)@T \in L_{EV}^+ \), and \( TR_1, \ldots, TR_k \) are transition rules. Each transition rule is of the form

\[ !EV_1, @f_1(T), \ldots, !EV_n, @f_n(T), +A_1, \ldots, +A_n, -A_n+1, \ldots, -A_k \leftarrow B_1, \ldots, B_m, C. \]

where \( n + \ell > 0, EV_1, \ldots, EV_n \in L_{EDB}^{ev}, m \geq 0, A_1, \ldots, A_n, A_{n+1}, \ldots, A_k \in L_{EDB}^{ev}, B_1, \ldots, B_m \in L_{EDB} \cup L_{DDB} \cup L_{IDB}, f_1, \ldots, f_n \) are temporal functions, and \( C \) is an optional choice atom in \{choice, choice\Any\}.

The informal semantics of an event activation rule is that, if the event \( e(X_1, \ldots, X_n) \) occurs at time \( t \in T \) and the body of the transition rule is evaluated true, then the facts \( A_1, \ldots, A_n \) are asserted at time \( t \), the facts \( A_{n+1}, \ldots, A_k \) are retracted at time \( t \), and the events \( EV_1, \ldots, EV_n \) are triggered to be executed at times \( f_1(t), \ldots, f_n(t) \).

**Definition 4.2** Let us assume a time domain \( T \) and a set of knowledge base EDB predicates are given. Then, a DATALOG\({}^{ev}\) program \( P^e = \{D-KB, EV\} \) over \( T \) and \( \sigma^{EDB} \) consists of a set D-KB of dynamic rules, called dynamic knowledge base, and a set EV of event activation rules. If the time domain is clear from the context, the program will be denoted simply by \( P \). The set of all DATALOG\({}^{ev}\)
programs whose dynamic knowledge base is stratified is denoted by $\text{DATALOG}^{\text{str}}$. 

If we are additionally given an extensional database $\text{EDB}$ encoding the (static) initial knowledge we are interested in, then we denote by $\mathcal{P}_{\text{pm}} \cup \{p \leftarrow | p \in \text{EDB}\}$, i.e., the program obtained by adding a fact for each atom in $\text{EDB}$. □

4.2 Modeling Dynamic Systems

We next illustrate the peculiarities of $\text{DATALOG}^{\text{str}}$ in modeling complex dynamic systems. Assume that you are the project manager of a software house and your clients ask you to implement novel tools. In order to supply each request, you might think at activating a project involving some employees. However, if you find that this is either not convenient for your company or not possible since you do not have employees enough to guarantee the actual implementation of the tool, you might also refuse the order. Figure 3 shows a possible workflow implementing the project management process. The process is quite complex because it involves several subtasks, each one activated by events triggered by previous tasks — denoted in the figure by labels over the arcs. We next present some guidelines on building a $\text{DATALOG}^{\text{str}}$ program $\mathcal{P}_{\text{pm}}$ for modeling this process.

The first event (init), causing the activation of the process, is an external one and occurs when an order is received. In fact, this event is responsible for the execution of the Receive Order task in which some preliminary activities, such as storing of client data, have to be performed. This can be modelled by means of an event activation rule of the form

$$[\text{init}(P) \cup (T)] \leftarrow$$

$$\text{receiveOrder}(P) \cup (T) \leftarrow$$

The task Receive Order notifies its completion through the activation of the event received, which in its turn activates the task Check Requirements:

$$[\text{received}(P) \cup (T)] \leftarrow$$

$$\text{checkRequirements}(P) \cup (T) \leftarrow$$

In case you find the order not convenient, the Check Requirement task notifies the notOK event to the refuse Order task:

$$[\text{notOK}(P) \cup (T)] \leftarrow$$

$$\text{refuseOrder}(P) \cup (T) \leftarrow$$

Otherwise, the event OK will lead to the acceptance of the order:

$$[\text{OK}(P) \cup (T)] \leftarrow$$

$$\text{acceptOrder}(P) \cup (T) \leftarrow$$

If the order is accepted, you will start the actual realization and you will eventually release the tool. Here, we consider just two phases: Requirement Analysis and Implementation. Notice that, in the latter task, it might happen that some employee leave the company, represented by means of the external event leave, possibly causing the failure of the project.

After the skeleton of the process is designed by means of the above event activation rules, $\text{DATALOG}^{\text{str}}$ can be used for modeling in simple and natural ways the different subtasks discussed above. In Figure 3, we have reported only some details for the Check Requirement task. Essentially, it comprises a check for the reliability of the client, which can be modelled as follow:

$$[\text{checkRequirement}(P) \cup (T)] \leftarrow$$

$$\text{verifyClient}(P) \cup (T) \leftarrow$$

The activity verifyClient is not further described here. We just assume that if the client is not reliable, it will eventually trigger the event notOK (causing the rejection of the order); otherwise it will trigger the event verifyStaffing, that is responsible for the staffing of employees in the project:

$$[\text{verifyStaffing}(P) \cup (T)] \leftarrow$$

$$\text{staff}(P) \cup (T) \leftarrow$$

Staffing a project is another complex task, which is executed into a new conceptual dimension as described next. Each project $p_i$ requires some specific skills, and thus the company has to assign a suitable set of employees to the project, such that all the required skills are granted to $p_i$. Of course, once an employee is assigned to a project team, she is not available for another project until the current one has been ended. Then the staffing must be also planned, since the management has to make a sequence of choices for selecting the employees to be included in the team. However, at the level of projects execution times, the details about such sequences of choices are not relevant, and hence it should be done in a conceptual finer dimension. Rather, it is crucial that this process is correctly performed, according to the following constraints: (i) an employee can be assigned to one project at a time, (ii) for each skill
required in a project, one employee must be present in the working team. A project is "staffed" if both the above conditions are satisfied.

We next define some DATALOG^{\textit{ev}} rules encoding our staffing problem and give just a flavor of its meaning, as the semantics of the language is formally presented in the next section. Figure 4 shows a possible extensional database EDB encoding the information about projects and employees in our project management example. For instance, we can see that project \( p_1 \) must be completed within four time units and requires a developer and a consultant, that employee \( e_1 \) has the skill \( s_1 \) (developer), etc. Then, \( \mathcal{P}_{pm} \) contains a set of DDB facts \( \text{inTeam}(\text{Project}\#, \text{Employees}\#)/t \) to encode the fact that an employee is enrolled in a project at some time \( t \). The following D-KB rules determine whether a project \( P \) is staffed at a time \( T \).

- \( \text{staffed}(P) \leftarrow \text{project}(P, \_), \neg \text{missingSkill}(P). \)
- \( \text{missingSkill}(P) \leftarrow \text{requiredSkill}(P, \_), \neg \text{skillInP}(P, \_). \)
- \( \text{skillInP}(S, P) \leftarrow \text{inTeam}(P, E), \text{employee}(E, \_). \)

Moreover, the program also contains the following rules for inferring the employees that are not currently involved into any project, and whose skills are still missing.

- \( \text{candidate}(E, P) \leftarrow \text{employee}(E, S), \neg \text{inSomeTeam}(E), \text{requiredSkill}(P, S, \_), \neg \text{skillInP}(S, P). \)
- \( \text{inSomeTeam}(E) \leftarrow \text{project}(P, \_), \text{inTeam}(P, E). \)
- \( \text{moreCandidates}(P) \leftarrow \text{candidate}(E, P). \)

The only event transition rule for the staffing problem has been already shown in Figure 3. Roughly speaking, the \textit{staff} event is responsible for choosing \textit{non-deterministically} an employee for the inclusion in the project having a skill that is still missing in the staffing. When the project is staffed the event \( \text{OK} \) is eventually triggered. Otherwise, i.e., if there are no more candidates and the project is not yet staffed, then \( \text{notOK} \) is triggered.

Finally, at the end of the project we release the team members (note that, in this case, we do not choose a particular employee, and in fact we want to delete all the employees in the team). This way, these employees may be enrolled in a further project.

\[
\text{[end}(P)/@T]\]
\[
\neg \text{inTeam}(P, E) \leftarrow \text{inTeam}(P, E). \]

It is worthwhile noting that, as several projects may be initialized, the staffing activities can be thought as concurrent processes that share the same resources (the employees). It follows that, in general, there exists bad combinations of choices such that some project may not be staffed.

Since the incomes are mainly due to the ability of properly managing the projects, you might think at developing a simulation environment that can help your decision. DATALOG^{\textit{ev}} might help your job by means of its powerful query language described in Section 6. In fact, after the process is modelled, as we shall show, you may query the program for verifying whether there exists a proper staffing that guarantees the satisfaction of the whole order (and, obviously, for computing such a \textit{plan}). Similarly, you may also identify those employees which are crucial for the implementation of a tool, i.e., the employees that will cause a failure of the project if they leave the company.

5. SEMANTICS

Let \( \mathcal{P}_{\text{EDB}} = (\text{D-KB}, \text{EV}) \) be a DATALOG^{\textit{ev}} program, over a time domain \( T \) and a knowledge base EDB. As usual, the Herbrand Universe \( \mathcal{U}^{\text{EDB}} \) of a \( \mathcal{P}_{\text{EDB}} \) is the set of all constants appearing in \( \mathcal{P}_{\text{EDB}} \).

\[
\begin{array}{|c|c|c|}
\hline
\text{Project} & \text{Duration} & \text{Employee} & \text{Skill} \\
\hline
p_1 & 4 & e_1 & s_1 \\
p_2 & 5 & e_1 & s_1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\# & \text{Skill} & \text{Description} \\
\hline
1 & s_1 & \text{developer} \\
2 & s_2 & \text{consultant} \\
3 & s_3 & \text{researcher} \\
\hline
\end{array}
\]

Figure 4: An extensional database EDB for the program \( \mathcal{P}_{pm} \).

A dynamic literal (resp., an event) in \( \mathcal{L}_{\text{EDB}} \cup \mathcal{L}_{\text{IDB}} \) (resp., in \( \mathcal{L}_{\text{EV}} \)) is ground if no variable occurs in it. The EDB (resp., IDB, EDB, EV) Herbrand Base, denoted by \( \mathcal{B}_{\text{EDB}} \) (resp., \( \mathcal{B}_{\text{IDB}}, \mathcal{B}_{\text{EDB}}, \mathcal{B}_{\text{EV}} \)), is the set of all ground extensional (resp., dynamic fact, intensional, event) literals that can be constructed with the predicate symbols in \( \sigma_{\text{EDB}} \) (resp., \( \sigma_{\text{IDB}}, \sigma_{\text{EDB}}, \sigma_{\text{EV}} \)), by replacing the variables in \( \sigma_{\text{vars}} \) by constants in the Herbrand universe and the time variables in \( \sigma_{\text{time}, \text{vars}} \) by time instances in \( T \).

Definition 5.1 An interpretation for the program \( \mathcal{P}_{\text{EDB}} \) consists of a pair \((S, E)\), where \( S \) is a set of ground literals and \( E \) is a set of ground events, such that

\[
S \subseteq \mathcal{B}_{\text{EDB}} \cup \mathcal{B}_{\text{IDB}} \cup \mathcal{B}_{\text{EV}} \cup \mathcal{B}_{\text{DB}} \\
E \subseteq \mathcal{B}_{\text{EV}}
\]

The minimum temporal argument occurring in the events in \( E \) is denoted by \( \text{nextTime}(I) \), while the maximum temporal argument occurring in the predicates in \( S \) is denoted by \( \text{curTime}(I) \). Finally, an interpretation \( I \) is \textit{feasible} if \( E = \emptyset \) or \( \text{curTime}(I) < \text{nextTime}(I) \).

Intuitively, a feasible interpretation \( I \) determines a truth value for all the predicates preceding the time \( \text{curTime}(I) \), and contains the information on the events that are currently triggered to occur in the future. In particular, a ground \( \text{IDB} \) or \( \text{EDB} \) predicate is true w.r.t. \( I \) if it is an element of it; a dynamic ground fact \( p@t \) is true w.r.t. \( I \) if there exists an element \( p@t' \in I \) such that \( t' \leq t \), and there is no literal \( \neg p@t'' \in I \) such that \( t'' < t' \).

Note that in the above definition, we assume that any \( \text{IDB} \) predicate asserted at a given time, remains valid till it is explicitly retracted from the database; indeed, the behavior of the \( \text{IDB} \) predicates is essentially \textit{inertial}, while the truth value of the \( \text{IDB} \) predicates must be determined at each time instant.

Finally, the special choice literals are defined to be always true w.r.t. to any possible interpretation \( I \), regardless whether they occur or not in \( I \).

We say that a ground transition rule \( tr \) is \textit{enabled} if all the literals occurring in the body of \( tr \) are true with respect to \( I \).

Example 5.2 Let us consider again the project management planning problem. Then,

\[
I_1 = \{ \{\text{staffable}(p_1, e_1)/3, \text{inTeam}(p_2, e_2)/@5\}, \{\text{end}(p_1)/@8\} \}
\]

and

\[
I_2 = \{ \{\text{inTeam}(p_2, e_3)/@6, \neg \text{inTeam}(p_2, e_4)/@5\}, \{\text{end}(p_2)/@2, \text{end}(p_1)/@9\} \}
\]

are both interpretations. However, the latter is not feasible since \( \text{nextTime}(I_2) = 2 \) and \( \text{curTime}(I_2) = 8 \). Moreover, note that in the former interpretation the predicate \( \text{inTeam}(p_2, e_3) \) is true in
Every time instant following 5, since it has been never retracted after its assertion at time 5.

Given an interpretation $I = \langle S, E \rangle$, we denote by $\text{triggered}(I)$ the set of all events in $E$ having temporal argument $\text{nextTime}(I)$, in the case $\text{nextTime}(I) \in T$; otherwise, we let $\text{triggered}(I) = \emptyset$. Let $\text{TR}(I)$ be the set of all transition rules such that all their activating events belong to $\text{triggered}(I)$, and $C(I)$ be the set of all choice predicates occurring in the rules in $\text{TR}(I)$. Moreover, let $\text{ground}_TR(I)$ be the set of all the ground instantiations $R$ of the rules in $\text{TR}(I)$ such that (i) all transition rules in $R$ are enabled, and (ii) the functional dependencies determined by the choice constructs in $C(I)$ are satisfied by $R$. Thus, $\text{ground}_TR(I)$ contains a set of enabled ground rules (coming from instantiations of the rules in $\text{TR}(I)$) for each possible way of enforcing the functional dependencies determined by the choices in $C(I)$.

Let $\text{chosen}_tr$ be any set of ground rules in $\text{ground}_TR(I)$. We denote by $A_I(\text{chosen}_tr)$ the set of all the dynamic atoms $p$ such that $\top p$ occurs in the head of some transition rule in $\text{chosen}_tr$ and $p$ is false w.r.t. $I$. Such a dynamic atom $p$ is said to be asserted. Similarly, $R_I(\text{chosen}_tr)$ is the set of all the dynamic literals $\lnot p$ such that $\bot p$ occurs in the head of some transition rule in $\text{chosen}_tr$ and $p$ is true w.r.t. $I$. In this case, we say that $p$ has been retracted. Finally, $\mathcal{E}_I(\text{chosen}_tr)$ is the set of the events triggered by all transition rules $r$ in $\text{chosen}_tr$ such that at least one dynamic atom is either asserted or retracted because of $r$.

In the sequel, the set of all the interpretations of a given program $\mathcal{P}$ is denoted by $\mathcal{I}_P$, while the set of all the subsets of $\mathcal{I}_P$ is denoted by $2^{\mathcal{I}_P}$.

**Definition 5.3** Let $\mathcal{P} = \langle D, KB, E \rangle$ be an event choice Datalog program. Then, we define $T : 2^{2^P} \rightarrow 2^{2^P}$ to be the function that, given a set of interpretations $\mathcal{I}$, outputs a set of interpretations $T(\mathcal{I})$ containing, for any $I = \langle S, E \rangle \in \mathcal{I}$ and any set of transition rules $\text{chosen}_tr \in \text{ground}_TR(I)$, all interpretations $(S', E')$ such that

\[
S' \in \text{SM}(D, KB) \cup S \cup A_I(\text{chosen}_tr) \cup R_I(\text{chosen}_tr) \cup \text{triggered}(E), \quad \text{and} \quad E' = E \cup \mathcal{E}_I(\text{chosen}_tr) \cup \lnot \text{triggered}(E).
\]

Note that, for any given interpretation $I = \langle S, E \rangle$, this function computes the set of all possible interpretations that can be obtained by triggering events and by asserting or retracting predicates, according to $I$. Note that any output interpretation $I' = \langle S', E' \rangle$ takes into account the consequences of the events triggered at the time $\text{nextTime}(I)$. All these events are removed from the set $S'$, while new events possibly planned to occur in the future are added to $E'$ through the set $\mathcal{E}_I(\text{chosen}_tr)$. The set $S'$ is any stable model of the dynamic knowledge base $D, KB$ evaluated over $S$ plus the asserted and retracted predicates, and including the recently occurred events, too.

We point out that, as a consequence of the non-deterministic choice constructs, the output of $T$ applied on a singleton $\{I\}$ is in general a set of multiple alternative interpretations, even in the case the dynamic knowledge base is stratified ($\text{DATALOG}_{\text{w}}^{\text{w}}$) program. However, it deterministically outputs a unique interpretation (for the given $I$) if the program is stratified and there are no “active” choices, i.e., $C(I) = \emptyset$.

**Definition 5.4** Let $\mathcal{P}$ be a $\text{DATALOG}_{\text{w}}^{\text{w}}$ program, $EDB$ be an input database, and $H$ a list of ground events, also called list of envisioned events. The evolution of the program $\mathcal{P}$ given $EDB$ and $H$ (short: the evolution of $\mathcal{P}_{EDB,H}$) is the succession of sets of interpretations $\hat{T}$ such that (i) $T_0 = \{\langle EDB, H \rangle\}$, and (ii) $T_{i+1} = T(T_i)$. For every $j > 0$, any interpretation $M \in \hat{T}_j$ is called a temporal model for $\mathcal{P}_{EDB,H}$.

Note that the definition of temporal model refers to a list $H$ of envisioned events, containing the events that are deterministically known to happen. Thus, $H$ can be used for simulating the actual behavior of a system modelled with $\text{DATALOG}_{\text{w}}^{\text{w}}$. For instance, in our running example the list $[\text{init}(p_1)@0, \text{init}(p_2)@2]$ is used for simulating a scenario in which two projects are going to be staffed at times 0 and 2, respectively. Under an abstract perspective, the events in $H$ are used for constraining the evolution of the $\text{DATALOG}_{\text{w}}^{\text{w}}$ program.

**Definition 5.5** Let $\mathcal{P}$ be a $\text{DATALOG}_{\text{w}}^{\text{w}}$ program, $EDB$ be an input database, and $H$ a list of ground events. A temporal model $M$ for $\mathcal{P}_{EDB,H}$ is a stationary model (for $\mathcal{P}_{EDB,H}$) if it is a fixpoint of $T$, i.e., if $M \in \hat{T}(\{M\})$. Then, $\text{curTime}(M)$ is called the converging time of $M$.

Another characterization of stationary models is provided by the following result.

**Proposition 5.6** A temporal model $\langle S, E \rangle$ for any program $\mathcal{P}_{EDB,H}$ is stationary if and only if $E = \emptyset$.

**Example 5.7** Assume that the project $p_1$ should start at the time instant 0 and the project $p_2$ at the time instant 2. This is encoded through the list of envisioned events $H = [\text{init}(p_1)@0, \text{init}(p_2)@2]$. Then, the graph reported in Figure 5 shows (the relevant atoms of) a temporal model of $\mathcal{P}_{\text{min}}$, given the extensional database $EDB$ in Figure 4 and the list $H$, where the project $p_1$ is staffed at time 4, whereas project $p_2$ is staffed during the time instant 6. Note that during time 6, a number of elementary steps are executed in the second time dimension (at instants (6, 1) and (6, 2)) for choosing the employees to be enrolled in $p_2$, namely, $e_1$ and $e_3$.

Note that this temporal model is also stationary as no further events are triggered to occur after the completion of the projects. Thus, its converging time is 12, when the last project ends.
Proposition 5.8 Let $P$ be a $\text{DATALOG}^{ev}$ program, $\text{EDB}$ be an input database, $H$ a list of ground events, and $I$ be an interpretation of $P$. Then, checking whether $I$ is a temporal model, as well as checking whether $I$ is a stationary model are polynomial time tasks.

Proof Sketch. All the non-deterministic issues can be solved by considering the actual values in the given interpretation $I$, both in the computation of stable models in the case of unstratified programs, and in enforcing the functional dependencies according to the choice constructs.

The set of all the temporal (resp. stationary) models of a given program $P_{EHB}$ is denoted by $TM(P_{EHB})$ (resp. $TSM(P_{EHB})$).

6. QUERIES AND COMPLEXITY ISSUES

Let us now describe how to query a $\text{DATALOG}^{ev}$ program about its possible evolutions on the basis of a envisioned future events.

$\text{DATALOG}^{ev}$ queries are formulated involving literals and special temporal quantifiers, as defined inductively below. Let $t$ be a time instant and $C$ a nonempty conjunction of ground literals having time arguments at most $t$ (w.r.t. the $\leq$ ordering). Then, $\forall \leq t C$ and $\exists \leq t C$ are queries. Moreover, let $t_1$ and $t_2$ be two time instants such that $t_1 < t_2$, let $C$ be a (possibly empty) conjunction of ground literals with time arguments at most $t_1$, and let $Q$ be a query, whose first quantifier is either $\exists \leq t_2$ or $\forall \leq t_2$. Then, $\forall \leq t_1 C \land Q$ and $\exists \leq t_1 C \land Q$ are queries.

A query $Q$ starting with an existential (resp., universal) quantifier is called an existential (resp., universal) query. Moreover, if the maximum number of nested quantifiers alternations in $Q$ is $k$, then it is called a $k$-existential (resp., $k$-universal) query.

Hereafter, given a model $M = \langle S, E \rangle$ for a $\text{DATALOG}^{ev}$ program, and a time $t$, we denote by $M \circ t = \langle S', E' \rangle$ the interpretation consisting of all the atoms having any temporal argument $t' \leq t$.

Given two temporal models $M$ and $N$, we say that $N$ is an evolution of $M$ from time $t$ if $M \circ t = N \circ t$. The set of all the evolutions of $M$ is denoted by $\text{evols}(M)$.

Let $M$ be a set of temporal models for a $\text{DATALOG}^{ev}$ program $P$. We say that a query $Q$ is true with respect to $M$ if one of the following conditions hold:

- $Q = \exists \leq t C$, where $C$ is a nonempty conjunction of ground literals, and there exists $M \in M$ s.t. all the literals in $C$ are true w.r.t. $M$; or
- $Q = \forall \leq t C$, where $C$ is a nonempty conjunction of ground literals and, for all $M \in M$, all the literals in $C$ are true w.r.t. $M$; or
- $Q = \exists \leq t (C \land Q')$, where $C$ is a (possibly empty) conjunction of ground literals, and there exists $M \in M$ s.t. all the literals in $C$ are true w.r.t. $M$ and $Q'$ is true w.r.t. $\text{evols}(M)$; or
- $Q = \forall \leq t (C \land Q')$, where $C$ is a (possibly empty) conjunction of ground literals and, for all $M \in M$, all the literals in $C$ are true w.r.t. $M$ and $Q'$ is true w.r.t. $\text{evols}(M)$.

Otherwise, we say that $Q$ is false w.r.t. $M$.

Definition 6.1 (Query answers) Let $P$ be a $\text{DATALOG}^{ev}$ program, $T$ a time domain, $\text{EDB}$ an extensional database, $H$ a list of envisioned events, and $Q$ a query. The answer of $Q$ over the program $P$ w.r.t. to $T$, given EDB and $H$, denoted by $Q(P_{EHB})$ is true (resp., stationarily true) if $Q$ is true w.r.t. the set of temporal models $TM(P_{EHB})$ (resp., of stationary models $TSM(P_{EHB})$).

The following example shows how to query $\text{DATALOG}^{ev}$ programs looking for suitable evolutions that meet some desired requirements.

Example 6.2 Consider again the project management planning program $P_{pm}$, the extensional database of Figure 4, and the list of envisioned events $H = \{\text{init}(p_1) \land \text{init}(p_2) \land \text{init}(p_3)\}$. Then, consider the query $Q_1 = \exists \leq \omega (\text{end}(p_1) \land \exists \leq \omega \text{end}(p_2))$. This query is true over the staffing program, given EDB and $H$, iff it is possible to staff and start the projects according to the given list of envisioned events, in such a way that there exists a temporal model where $p_1$ is completed within time 9, and there is an evolution of this model where $p_2$ is completed within time 12. In our case, $Q_1$ is in fact true, as witnessed by the temporal model $M_1$ shown in Figure 6.a. In this model, during the gray time instants, the staffing processes occur, while during the black time instants the projects are executed. Note that each gray time instant may involve a number of elementary steps that are executed in different time instants of the second (finer) time dimension.

Theorem 6.3 (Stationary model existence) Deciding whether $P_{EHB}$ has a stationary model is PSPACE-complete. Hardness holds even if $P$ is stratified.

It is worthwhile noting that the problem is much more easier, if we are satisfied with any temporal model, rather than requiring that the model is stationary.
Event Calculus lies in the ability of both defining external events, that also considers indeterminacy in the occurrence of events. Our approach is designed to work at a low level of details, by taking the capability of choice to express nondeterminism.

The main novelty w.r.t. to the above mentioned extensions of Event Calculus lies in the ability of both defining external events and modeling the non-deterministic effects of such events so that the actual validity of properties over the time dimensions are tested in a context of evolving knowledge bases. Indeed we face the knowledge representation problem with a different perspective: DATALOG\textsuperscript{ev} has been not designed for reasoning on maximal validity intervals, but rather for reasoning on the evolution of a given (logically) modelled domain in which the effects of the actions are not known in advance, as it is often the case in real applications. In fact, our language is closer to Dynamic Logic Programming, which extends logic programming with amenities for modeling and reasoning on evolving knowledge bases [2, 4, 3].

In this context, the rules of the program may be updated due to some events and hence modify the global state of the world. Different states sequentially ordered may represent time periods as in [3], that can be eventually combined with other dimensions, such as credibility of the sources and specificity of the updates [27]. In this field, we mention LUPS [4], whose core language is constituted by update commands (such as assert and retract) that can be also made conditional on the occurring of certain conditions by means of the clause when. Two extensions of LUPS, namely the specification of commands whose execution depends on other concurrent commands and the inclusion of external events, have been added into the language EPI [17]. In [2], it has been pointed out that the above mentioned languages do not adhere deeply at the LP doctrine, as they are too verbose and make use of many additional keywords. In order to provide a more declarative way for specifying updates in [2], it is proposed EVOLP, in which we may specify some rules updating the original program. Each time such rules are in the model of the program, the assertion are done, a new program is computed, and the process continues.

Theorem 6.5 (Query answering under temporal models) If the query $Q$ is $k$-existential (resp., $k$-universal), deciding whether the answer $Q(P^T_{EBH})$ is true is $\Sigma_k^P$-complete (resp. $\Pi_k^P$-complete). Hardness holds even if $P$ is stratified.

Interestingly, query answering under stationary models is not more difficult then deciding the existence of a stationary model.

Theorem 6.6 (Query answering under stationary models)

Deciding whether the answer $Q(P^{T}_{EBH})$ is stationary true is PSPACE-complete.

7. RELATED WORK AND CONCLUSION

We have presented an extension of DATALOG with events and choice, called DATALOG\textsuperscript{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 us to model a number of alternative potential evolutions of a program by means of dynamic rules that assert both dynamic facts and actions (i.e., triggered events), using the capability of choice to express nondeterminism.

A distinguished feature of this language is the ability of handling multiple time dimensions. Time granularities has been first introduced in Event Calculus (EC). Given a set of event occurrences, EC derives the maximal validity interval (MVIs) over which some properties hold. The event occurrence as well as the relationship between events and properties are specified by means of suitable clauses, and, in fact, a declarative specification of the derivation of MVIs can be straightforwardly obtained in PROLOG. The approach in [34] extended the single timeline of EC into a totally ordered set of different timelines $\{T_1, \ldots, T_n\}$, such that each $T_i$ is of a finer granularity than $T_{i-1}$. Similar ideas have been applied in [11], that also considers indeterminacy in the occurrence of events (for this latter aspect see also [18, 10]).

The main novelty w.r.t. to the above mentioned extensions of Event Calculus lies in the ability of both defining external events and modeling the non-deterministic effects of such events so that the actual validity of properties over the time dimensions are tested in a context of evolving knowledge bases. Indeed we face the knowledge representation problem with a different perspective: DATALOG\textsuperscript{ev} has been not designed for reasoning on maximal validity intervals, but rather for reasoning on the evolution of a given (logically) modelled domain in which the effects of the actions are not known in advance, as it is often the case in real applications. In fact, our language is closer to Dynamic Logic Programming, which extends logic programming with amenities for modeling and reasoning on evolving knowledge bases [2, 4, 3].

In this context, the rules of the program may be updated due to some events and hence modify the global state of the world. Different states sequentially ordered may represent time periods as in [3], that can be eventually combined with other dimensions, such as credibility of the sources and specificity of the updates [27]. In this field, we mention LUPS [4], whose core language is constituted by update commands (such as assert and retract) that can be also made conditional on the occurring of certain conditions by means of the clause when. Two extensions of LUPS, namely the specification of commands whose execution depends on other concurrent commands and the inclusion of external events, have been added into the language EPI [17]. In [2], it has been pointed out that the above mentioned languages do not adhere deeply at the LP doctrine, as they are too verbose and make use of many additional keywords. In order to provide a more declarative way for specifying updates in [2], it is proposed EVOLP, in which we may specify some rules updating the original program. Each time such rules are in the model of the program, the assertion are done, a new program is computed, and the process continues.

Besides the points in common with DATALOG\textsuperscript{ev}, we stress two important differences: (i) The paradigm of multidimensional updates cannot easily fit the need of representing multiple time dimensions; in fact, the above mentioned works assume only one temporal dimension within a single granularity, while the other dimensions refers to additional properties of the world. (ii) Languages such as EVELOP are not query driven, in the sense that there is no notion of finding updates to satisfy a query. An EVELOP program is concerned with finding the meaning of a given KB after a succession of updates. Conversely, DATALOG\textsuperscript{ev} has been specifically designed for being queried in order to perform temporal reasoning.

When comparing DATALOG\textsuperscript{ev} with the growing body of the proposals of action languages, we need to emphasize that our language is able to model a static knowledge as well as a dynamic one and, hence, is able to model actions with both direct and indirect effects, covering the main features of languages A, B (see, for a comparison, [21] and AC [42]). It also provides a set of primitives for reasoning on past events, thus capturing the power of Past Temporal A. Moreover, it deals with concurrent actions such as languages C [22] and AC [6], and it enables to reason about actual and hypothetical occurrences of concurrent and non-deterministic actions, such as language $L_2$ [7]. Moreover, a distinguishing feature w.r.t. actual action languages, is the ability of reasoning at any level of details. As pointed out by Baral [5], situation and event calculi, are close in their spirit as they aims at modeling a changing environment at a high level of detail, while the temporal reasoning approach is designed to work at a low level of details, by taking care of many aspects (e.g., the actual time of the occurrence of the events) besides the rough effect of actions.

Depending on the user needs, DATALOG\textsuperscript{ev} can be used as a framework for abstract reasoning on situations, but it is also able to plan (at the desired level of details) the actions to perform for achieving a given temporal goal. Hence, the language shares the same perspective of [5].

We conclude by pointing out that DATALOG\textsuperscript{ev} is essentially an extension of DATALOG and hence should be compared to the different proposals of extending logic programming with temporal logics (see [36], for a survey of the different proposals). Among the first proposals, we recall Datalog\textsuperscript{Ev} [13] and Templog [1]. The former is DATALOG extended with one successor modeling the advance of the time, while the latter is an extension that allows a restricted use of modal temporal operators. Despite their different syntax (Datalog\textsuperscript{Ev} seems an immediate extension of DATALOG), it has been proved that they are equivalent as for expressiveness and completeness. Starlog [14] is another logic language that adds an additional time-argument to every PROLOG predicate, and is designed for general purpose programming, for simulation, and for
modeling reactive systems. We point out that none of the above approaches deals with multiple time dimensions, and with complex temporal functions besides classical modal operators. Moreover, more importantly, none of the above approaches deal with external or internal events and with non-deterministic effect of actions. Hence, they appear not suited for modeling complex situations where a knowledge base is updated due to the nondeterministic occurrence of actions.

Finally, it is worth mentioning that we are implementing DATALOG by exploiting the disjunctive Datalog system dlv [28] as its engine core. This implementation is based on an extension of the notion of XY-stratification previously used to model update and active rules [45].

8. REFERENCES


