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

G. Greco, A. Guzzo, Saccà D., F. Scarcello

RT-ICAR-CS-04-8 **Maggio 2004**

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

On the Mining of the Complex Workflow Schemas

G. Greco[1], A. Guzzo[1], F. Scarcello[1], D. Saccà[1,2]

Rapporto Tecnico N.:
RT-ICAR-CS-04-8

Data:
Maggio 2004

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, Via P. Bucci 41C, 87036 Rende(CS)
[2] Università degli Studi della Calabria, DEIS, Via P. Bucci 41C, Rende (CS)

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
DEIS ICAR-CNR
Università della Calabria
87030 Rende - Italy
sacca@unical.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$^{!ev}$ 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 temporal dimensions in order to model a finer control of evolution. A comprehensive study of the computational complexity of answering DATALOG$^{!ev}$ queries is reported.

1. INTRODUCTION

Finding a suitable declarative framework for modelling 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., [18, 33]) 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 ([27]), temporal reasoning (see, e.g., [30]) and event calculus ([23]). Several recent proposals exploit, instead, logic programming and, specifically, the answer set paradigm for developing domain-independent planning languages [?]. Besides the very declarative modelling features of logic programming, the most interesting aspects 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 XSB [?], GnT [?], DLV [?], *Smodels* [?], DeReS [?], and ASSAT [?]. The language $\mathcal{K}$ [15] is a prototypical representative of the languages exploiting such an approach. In fact, it is completely based on the principles and methods of logic programming and its main feature is the ability of dealing with incomplete knowledge, i.e., of modelling scenarios in which the designer has a partial knowledge of the world, only. Moreover, it has been actually implemented as a front-end for the DLV system [].

One of the major limitation of the language $\mathcal{K}$ as well as of most of the other logic-based languages for reasoning about action is the lack of an explicitly support for time in the planning. Moreover, the few logic-languages dealing with timed actions [?] rarely consider the possibility of dealing with multiple time units, and, besides, such proposals mainly consist in finding 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 explode the time into a number of dimensions at different scales in order to model the validity of properties over coarser or finer time intervals. We stress 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. Multiple time units can be profitably used for two main purposes:

- ▷ They can be used for modelling in a more realistic way a given planning problem. In fact, recent research has recognized that systems supporting multiple granularities of time and reasoning involving these units are two important issues, as all the human activities are essentially related to clock units, such as weeks, days, and hours (an overview of different proposal and a definition of a unifying model for time granularities is presented in [?]).
- ▷ They can be used for dealing with plans at different level of details. In fact, each time dimension might be seen as a conceptual dimensions; so we can employ a main dimension for reasoning about some complex activities, and one auxiliary time dimension for modelling 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, as the following example shows.

We believe that both the above applications might be of interests in practical contexts. In fact, as for the first issue a language supporting multiple time units will increase its knowledge representation power, while for the second issue it becomes also a viable way for Hierarchical Task Network (HTN) planning (Sacerdoti 1974). HTN is an approach to planning where problem-specific knowledge is used to remedy the computational intractability of classical planning. This knowledge is in the form of task decomposition directives, i.e. the planner is given a set of methods that tell it how a high-level task can be decomposed into lower-level tasks. The

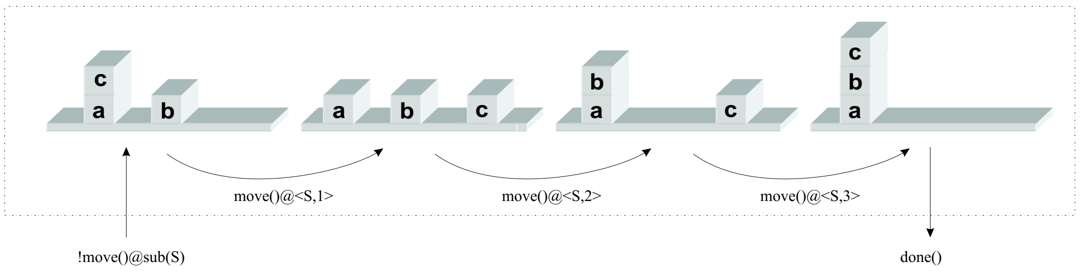


Figure 1: Blocksworld planning problem.

HTN planning problem consists in computing a sequence of primitive tasks that corresponds to performing the initial set of high-level tasks. First attempts to define languages able to explicitly deal with time and HTN planning problems have been done in [**?**], designing suitable extension of the Golog/ConGolog (Levesque et al. 1997; De Giacomo, Lesperance, Levesque 2000) languages (based on situation calculus).

In this paper we tackle the above knowledge representation issues in the logic programming context, and we propose a new language, called Event Choice DATALOG (DATALOG$^{!ev}$ for short), for modelling and querying action theories with different time granularities. The language can be used for defining declaratively subtasks that may be eventually combined in order to solve complex activities, in a way that is completely transparent to the user. We point out that the recombination of pieces of programs is not natural in logic programming context, in which the union of two programs may have unexpected semantics. Conversely, one interesting peculiarity of DATALOG$^{!ev}$ is its *modularity* that constitutes the basis for its applicability in HTN planning contexts. In fact, any program dealing with some simple planning problem can be reused in more complex systems by allowing it to work in a proper subunit of time without any interference with previously defined modules. Indeed, all the reasoning activities in subunits are seen instantaneously as far as the main unit is concerned. The language achieves this flexibility by exploiting two basic constructs for defining planning problems.

- **Event Activation Rules:** The language models transition among the possible states of the world, in the same spirit of $\mathcal{C}$ [20], by exploiting the notion of event. Indeed, the occurrence of an event might cause the application of a rule that modifies the state by asserting or retracting some facts (*fluents*) and that triggers other events in its turn. Besides the internally triggered events, the language also support the interaction with external ones. This latter features is of particular interest for simulating and reasoning about possible scenarios, as we shall describe in the following.
- **Choice construct:** 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, DATALOG$^{!ev}$ allows only stratified negation, but its rules may contain *choice* constructs, that provide nondeterministic features. In particular, if we are not interested in a particular outcome (temporal evolution) of the program, the *choice* is able to model the so called don't-care nondeterminism.

We will show that DATALOG$^{!ev}$ is well suited for modelling and reasoning about complex dynamic systems in real applicative scenarios, thereby being a powerful run-time environment for their **simulation**. To this aim DATALOG$^{!ev}$ is equipped with a powerful querying mechanism. Since the same event may actually occurs in different forms, because of the nondeterministic nature of transition rules, given a list of envisioned events, the problem consists in verifying whether there exist particular sequences of further event occurrences that eventually satisfy a given goal, and in returning a possible future state satisfying this goal.

1.1 Overview of the Language

In a nutshell, DATALOG$^{!ev}$ is a language for modelling the evolution of knowledge states, triggered by events and guided by nondeterministic transition rules. It combines the capability of the *choice* construct to express nondeterminism with the *event activation rules*, used for modelling events occurring at certain 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, i.e., it can be used for simulation and planning purposes. We next give a brief exposition of such features of the language by considering the *Blocksworld* planning problem [31]. The example will be next enriched in order to show how DATALOG$^{!ev}$ can deal with HTN planning problems as well.

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 `a` and `b` are on the table, while block `c` is in the top of `a`. We can move a block at 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 `a` is on the table, `b` is on the top of `a`, and `c` on the top of `b` — see Figure 1.

The first component of a DATALOG$^{!ev}$ program used for modelling such program is the *background knowledge* expressed as a set of facts, denoted by `EDB` (extensional database), which are assumed to do not change over the time. These facts specifies the object involved in the modelled domain. In our example, `EDB` consists of the facts

```
block(a). block(b). block(c).
```

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

$$\texttt{on(a, table). on(b, table). on(c, a).}$$

The third component is constituted by a set of dynamic rules, denoted by D-KB (dynamic knowledge base). These rules are essentially datalog rules (with stratified negation) whose predicates are equipped with a time argument. Rules in 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 $\texttt{p(X)@T} \leftarrow \texttt{q(X)@T} - (2)$ imposes that predicate p(X) is true two instants of time after predicate q(X) is. In the special case that predicates in dynamic rules deals with the same time instant, the rules can be used for representing *static knowledge*, i.e., invariant over the time – in such cases, the time argument is often stripped off. For instance, in our running example, D-KB contains exactly the rules

$$\begin{array}{l}\texttt{fixed(B)} \leftarrow \texttt{on(B}', \texttt{B)}\ \texttt{block(B)}.\\ \texttt{goodLocation(D)} \leftarrow \texttt{block(D)},\ \neg\texttt{fixed(D)}.\\ \texttt{goodLocation(D)} \leftarrow \texttt{D} = \texttt{table}.\\ \texttt{done()} \leftarrow \texttt{on(a, table)},\ \texttt{on(b, a)},\ \texttt{on(c, a)}.\end{array}$$

Intuitively, fixed(B) is false at a given time T if the block B has no other block on the top of it, and, hence, if it can be freely moved. Then, goodLocation(D) is true if D is the table or a block on the top of which there are no other blocks. Moreover, 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*. This rules states that whenever a given event is (internally or externally) triggered, a set of actions will be performed. In our example, we only consider the event of requiring a move of a block.

$$\begin{array}{lll}[\texttt{move()@T}] & & \\ \quad !\texttt{move()@T}+(\texttt{10}), & & \\ \quad -\texttt{on(S, X)}, +\texttt{on(S, D)} & \leftarrow & \neg\texttt{done()},\\ & & \texttt{on(S, X)},\ \neg\texttt{fixed(S)},\\ & & \texttt{goodLocation(D)},\ \texttt{D} \neq \texttt{S},\\ & & \texttt{choiceAny()}.\end{array}$$

Intuitively, when move is triggered at time T we check for the condition in the body of the rule. Notice that predicates in the body of the rule do not have an explicit time argument; we will use such shorthand in the case the time arguments are the same of the one exploited in the invocation of the event, i.e., if we are looking at the state of the world simultaneously to the occurrence of an event. Specifically, in the above rule, we check whether the planning has been not yet completed (done() is false) and whether there is a block S that is not fixed that can be moved on the top of a goodLocation. Obviously, there are several possible choices, i.e., several blocks to be moved and several locations for each block. Then, the predicate choiceAny() is a directive of our language that ensures that only one of these possibilities is non-deterministically chosen. After the choice is done, the status of the world is updated in the head of the rule, that is on(S, X) is retracted and on(S, D) is asserted for denoting that S has been actually moved. Moreover, since we have not yet completed the planning, the event move() is internally triggered another time, in the next time unit (T+ (10)). Thus, we are assuming that each move requires ten time units, e.g., ten seconds.

The interesting feature of the above formalization is that the user had to write simple rules involving non-deterministic actions. The focus in writing DATALOG$^{!ev}$ programs goes only in properly defin-

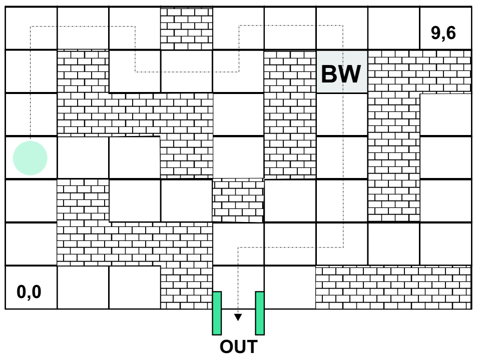


Figure 2: Labyrinth.

ing one step of the transition only. Then, the non-deterministic transitions ensure that all the possible evolutions will be equally considered and represent in a compact way all the possible sequences of moves (even the infinite, cyclic ones that do not lead to the final configuration). Roughly speaking, the choiceAny() construct ensures a form of don't-care non-determinism in which any evolution is admissible. Obviously, one is often interested in finding a particular sequence of moves that led to the achievement of the goal. This can be easily specified by means of the last component of DATALOG$^{!ev}$ i.e., its query language. In order to query a program, we have to specify a set of envision events that are the external events that are already known to happen. In our case, the list can be $\texttt{H} = [\texttt{move()@0}]$, specifying that we start the planning at time 0. Then, a query of the form $\texttt{Q} = \exists^{@\texttt{t}}\texttt{done()}$ will be true iff there exists a possible way for achieving the desired final condition within time t. As we can see from the figure, the query will be evaluated true for $\texttt{t} \geq 30$ (seconds). In the following, we shall see how the blockworld problem can be reused for solving a subgoal of a more complex planning problem. Then, the event move()0 will not constitute an external event, but will be internally triggered in the program encoding such a complex problem.

1.1.2 Multidimensional Planning

Let us now consider a more complex scenario. You have to devise a program that is able to control a robot in a labyrinth (the grid in Figure 2). Indeed, the robot is in the position $(0, 3)$ and must arrive in position $(4, 0)$. As far as this "main" problem is concerned, you are only interested in counting the number of steps in the robot's escaping path, so that you care only at the spatial dimensions — for instance, you might think at minimizing the length of the path.

However, while looking for the exit, the robot has to perform some other subtasks within some time bounds, depending on his position. For instance, when he is in position $(6, 5)$ he has to solve the blockworld problem in at most 30 seconds. Thus, as far as subtasks are concerned, you are interested in actual time dimensions. We stress that the scenario is prototypical of all such situation in which we have to realize some goal which involves the achievement of other partial goals. We shall next see that DATALOG$^{!ev}$ allow us to reuse in a quite simple and elegant way the program that models the blocksworld problem, and that it can easily deal with both the two dimensions of interests, i.e., the length of the path and the time bounds of the subtasks.

Let us preliminary solve the problem of escaping from the labyrinth without caring of the subtasks. The static knowledge consists of the possible directions for the walk, i.e.,

$$\mathtt{dir(n).\ dir(s).\ dir(w).\ dir(e).}$$

plus the facts determining the coordinates of the grid, i.e., $\mathtt{coordX(X)}$ for $0 \leq \mathtt{X} \leq 9$, and $\mathtt{coordY(Y)}$ for $0 \leq \mathtt{Y} \leq 6$, and the facts asserting the presence of wall in a given position, i.e., $\mathtt{wall(X,Y)}$.

The dynamic database DDB comprises pos defining the current position of the robot in the labyrinth (initially, we assert $\mathtt{pos(0,3)}$). Then, D-KB consists of the rules

$$\begin{array}{l}
\mathtt{arrived()} \leftarrow \mathtt{pos(4,0)}.\\
\mathtt{walk(D,X,Y+1)} \leftarrow \mathtt{D = n, pos(X,Y), Y < 6, \neg wall(X,Y+1)}.\\
\mathtt{walk(D,X,Y-1)} \leftarrow \mathtt{D = s, pos(X,Y), Y > 0. \neg wall(X,Y-1)}.\\
\mathtt{walk(D,X-1,Y)} \leftarrow \mathtt{D = w, pos(X,Y), X > 0. \neg wall(X-1,Y)}.\\
\mathtt{walk(D,X+1,Y)} \leftarrow \mathtt{D = e, pos(X,Y), X < 9. \neg wall(X+1,Y)}.
\end{array}$$

The first rule is used for determining whether the tasks of the robot has been accomplished. Predicate walk contains instead the next location of the robot after a walk — notice, that we check whether the next location is admissible, i.e., if it falls within the grid and if it is not in a place where a wall is.

Finally, in order to model the walk of the robot we exploit an event $\mathtt{pathFinder()@S}$, where S is the step of the path. This event activets the following rules

```
[pathFinder()@S]
    !pathFinder()@S++),
    −pos(X,Y),
    +pos(XN,YN) ← ¬arrived(),dir(D),
                  coordX(XN),coordY(YN),
                  walk(D,XN,YN),
                  pos(X,Y),
                  choiceAny().
```

The first rule checks whether the robot is not arrived at the exit. Then, it selects a new direction which can be pursued by the robot, i.e., such that $\mathtt{walk(D,XN,YN)}$ is true. The selection is performed non-deterministically, by means of the $\mathtt{choiceAny()}$ construct. After the selection is done, the position of the robot is updated and the event pathFinder is triggered for the successive step, denoted by $\mathtt{S{+}{+}}$ (shorthand for $\mathtt{S+(1)}$).

We want to stress one more time the very interesting way of defining $\mathtt{DATALOG}^{!\mathrm{ev}}$ programs, in which one has to specify only the general way a transition is carried out. Then, the event activation rules models all the possible evolutions, i.e., all the possible paths. Finally, if one is interested in a path leading to the exit it suffice to query the program with $\mathtt{Q} = \exists^{@\mathtt{t}}\mathtt{arrived()}$, which intuitively means that we are looking for an evolution (path) that will lead to the exit. We believe that this is a very interesting and intuitive way of specifying planning problems.

Let us consider now, the fact that when the robot is in position $(6,5)$, he has to solve the blocksworld subtask. This can be easily modelled by means of another event activation rule of the form

```
[pathFinder()@S]
    !move()@sub(S) ← pos(6,5).
```

The above rule essentially triggers the execution of the planning of the blocks in a lower dimension ($\mathtt{sub(S)}$) w.r.t. the one used for planning the path. Actually, this dimension is a temporal one as we have described in the previous section. What is relevant in our approach is that all the operations that are performed in such finer dimension are seen instantaneously as far as the higher dimension is concerned. It follows that in the sub-dimension we take care of the time, but in the main dimension we take care of the steps.

Then, the query $\mathtt{Q} = \exists^{@\langle \mathtt{s},30\rangle}\mathtt{done()} \wedge \exists^{@12}\mathtt{arrived()}$ is true if it is possible to find a path leading to the exit in 12 steps such that the blocksworld problem can be solved in 30 seconds at most.

1.2 Organization

The paper is organized as follows. We introduce the multidimensional time domain in Section 2, and in Section 3 we present the language $\mathtt{DATALOG}^{!\mathrm{ev}}$. We illustrate, in the subsequent section, its model theoretic semantics, by introducing the notion of *temporal* and *stationary model*. Next, in Section 5 we formulate queries on $\mathtt{DATALOG}^{!\mathrm{ev}}$ programs and analyze their complexities. Finally, in Section 6 we present related work, discuss the main novelties of our language, and draw our conclusions.

2. PRELIMINARIES ON DATALOG

A *Datalog program* $\mathcal{P}$ 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 $\mathcal{P}$ is denoted by $ground(\mathcal{P})$; the *Herbrand universe* and the *Herbrand base* of $\mathcal{P}$ are denoted by $U_{\mathcal{P}}$ and $B_{\mathcal{P}}$, respectively.

Let an interpretation $I \subseteq B_{\mathcal{P}}$ be given — with a little abuse of notation we sometimes see I as a set of facts. Given a predicate symbol r in $\mathcal{P}_D$, $I(r)$ denotes the relation $\{t : r(t) \in I\}$. Moreover, $pos(\mathcal{P}, I)$ denotes the positive logic program that is obtained from $ground(\mathcal{P})$ 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* [18] if $I = \mathbf{T}^{\infty}_{pos(\mathcal{P},I)}(\emptyset)$, i.e., it is the least fixpoint of the classical *immediate consequence transformation* for the positive program $pos(\mathcal{P}, I)$.

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 $\mathtt{DATALOG}$ programs is simply called $\mathtt{DATALOG}$; the subclass of all stratified programs is called $\mathtt{DATALOG}^{\neg_s}$.

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 [**?**], 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[1] 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 $X, Y \subseteq Z$ — 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 $choice((),(Z))$ 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 [21] and [8].

Each time *instant* is a tuple $\langle t_1, ..., t_n\rangle$, where n is the *current dimension* of this instant and each t_i is a natural number. The time $\langle 0\rangle$ 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.

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 $T^{\omega_1,m}$ this temporal domain.

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 modelling 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, as the following example shows.

All time domains T are linearly ordered according to the usual lexicographic 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 $\mathtt{t} = \langle t_1, ..., t_{n-1}, t_n\rangle$, we can apply one of the following operators, also called *temporal functions over* T:

- $\mathtt{t} + \mathtt{k}$, with $k \geq 0$ natural number, that increments (if possible) the time in the current dimension n of k units, i.e., it outputs $\langle t_1, ..., t_n + \mathtt{k}\rangle$; if the increment k is not possible, because of some bound on the current dimension n, then $\mathtt{t} + \mathtt{k}$ is undefined.
- $\mathtt{t} - \mathtt{k}$, with $k \geq 0$ natural number, that decrements (if possible) the time in the current dimension of k units, i.e., it outputs $\langle t_1, ..., t_n - \mathtt{k}\rangle$; if $t_n < k$, then $\mathtt{t} - \mathtt{k}$ is undefined.
- $\mathtt{sup}(\mathtt{t})$, which is defined if the current dimension n is greater than 1, and returns the time instant $\mathtt{sup}(\mathtt{t}) = \langle t_1, ..., t_{n-1} + 1\rangle$, i.e., it projects t onto the preceding dimension $n-1$ and increments the time in that dimension.
- $\mathtt{sub}(\mathtt{t})$, that outputs $\langle t_1, ..., t_{n-1}, t_n, 1\rangle$, i.e., it creates (if possible) a new time dimension; if n is the maximum allowed number of dimensions in T, $\mathtt{sub}(\mathtt{t})$ is undefined.
- t, i.e., the *identity* function.

Moreover, t++ and t-- are shorthand for $\mathtt{t} + 1$ and $\mathtt{t} - 1$, respectively. □

The above functions is all we need for reasoning about events with the language we present in this paper. Note however that many other complex temporal operators have been proposed for different purposes in the literature (e.g., the operators for converting time instants and scaling intervals [14]).

4. EVENT CHOICE DATALOG

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

4.1 Syntax

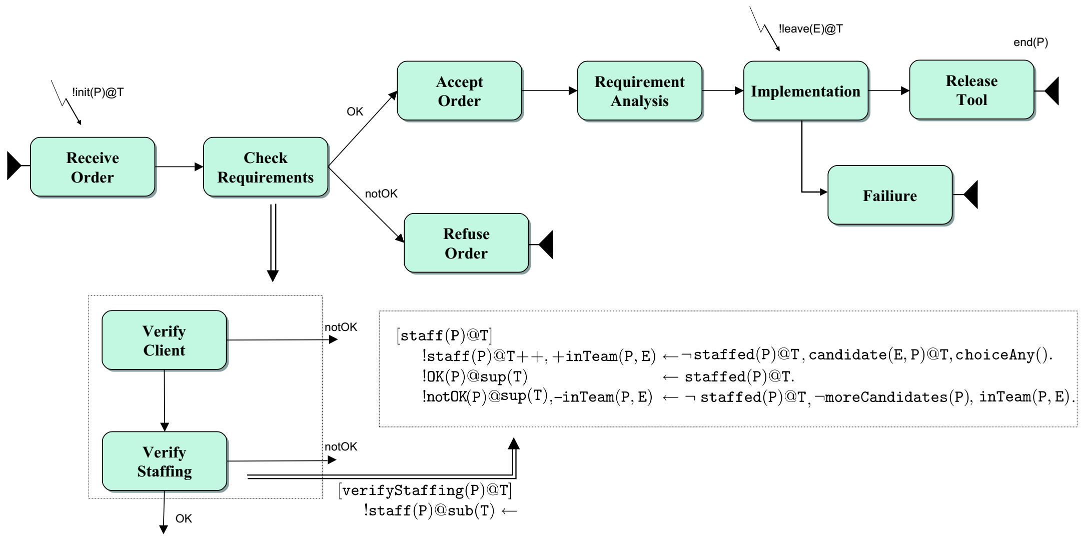
Roughly speaking, all $\mathtt{DATALOG}^{!ev}$ predicates are enriched with an additional argument that provides the time dimension: for any literal p and each time instant t, p@t is true if p holds at time t.

We assume that three sets of constants, variables, and time variables symbols, σ^{const}, σ^{vars}, and σ^{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 $\sigma^{\mathtt{EDB}}$, $\sigma^{\mathtt{DDB}}$, $\sigma^{\mathtt{IDB}}$, and $\sigma^{\mathtt{EV}}$ be disjoint sets of predicate symbols, with associated arity (≥ 0). Then, an EDB *atom* has a "classical" format $\mathtt{p}(\mathtt{s}_1, \ldots, \mathtt{s}_n)$ where p is a symbol in $\sigma^{\mathtt{EDB}}$ and $\mathtt{s}_1, \ldots, \mathtt{s}_n$ are terms. Instead DDB (dynamic extensional predicates), IDB (intensional predicates), and EV (event predicates) atoms are of the form $\mathtt{p}(\mathtt{s}_1, \ldots, \mathtt{s}_n)@\mathtt{f}(\mathtt{t})$, where p is a symbol in $\sigma^{\mathtt{DDB}}$, $\sigma^{\mathtt{IDB}}$, $\sigma^{\mathtt{EV}}$), respectively, n is the arity of p, $\mathtt{s}_1, \ldots \mathtt{s}_n$ are terms, f is a temporal function over the domain T, and t is a time instant or a time variable in σ^{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 $\mathcal{L}_{\mathtt{EDB}}$ (resp. $\mathcal{L}_{\mathtt{DDB}}$, $\mathcal{L}_{\mathtt{IDB}}$, $\mathcal{L}_{\mathtt{EV}}$). Furthermore, for any set of literals $\mathcal{L}$, $\mathcal{L}^+$ and $\mathcal{L}^-$ denote the sets of its positive and of its negative literals, respectively.

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



```
[staff(P)@T]
    !staff(P)@T++, +inTeam(P, E) ←¬staffed(P)@T, candidate(E, P)@T, choiceAny().
    !OK(P)@sup(T)                ← staffed(P)@T.
    !notOK(P)@sup(T),-inTeam(P, E) ← ¬ staffed(P)@T, ¬moreCandidates(P), inTeam(P, E).

[verifyStaffing(P)@T]
    !staff(P)@sub(T) ←
```

Figure 3: Project management.

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

$$[\mathtt{e(X_1, ..., X_n)@T}] \quad \mathtt{TR_1} \ \ldots \ \mathtt{TR_k}$$

where $\mathtt{e(X_1, ..., X_m)@T} \in \mathcal{L}^+_{\mathtt{EV}}$, and $\mathtt{TR_1, ..., TR_k}$ are *transition rules*. Each transition rule is of the form

$$\begin{array}{l} \mathtt{!EV_1@f_1(T), ..., !EV_n@f_n(T),} \\ \mathtt{+A_1, ..., +A_h, -A_{h+1}, ..., -A_\ell} \leftarrow B_1, ..., B_m,\ C. \end{array}$$

where $\mathtt{n} + \ell > 0$, $\mathtt{EV_1, ..., EV_n} \in \mathcal{L}^+_{\mathtt{EV}}$, $\mathtt{m} \geq 0$, $\mathtt{A_1, ..., A_h, A_{h+1}, ..., A_\ell} \in \mathcal{L}^+_{\mathtt{DDB}}$, $\mathtt{B_1, ..., B_m} \in \mathcal{L}_{\mathtt{EDB}} \cup \mathcal{L}_{\mathtt{DDB}} \cup \mathcal{L}_{\mathtt{IDB}}$, $\mathtt{f_1, .., f_n}$ are temporal functions, and $\mathtt{C}$ is an optional choice atom in $\{choice, choiceAny\}$. □

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

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

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

4.2 Modelling Dynamic Systems

We next illustrate the peculiarities of $\mathtt{DATALOG}^{!ev}$ in modelling complex dynamic systems. Assume that you are the project manager of a software house. Then, your clients ask you for implementing 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 not convenient for your company or not possible since, for instance, you do not have enough employees to guarantee the satisfaction of the requirement, you might also refute the order. Figure 3 shows a possible workflow implementing the process of accepting/refusig an order. The process is quite complex for it involving several subtasks: after the order is received (by means of the event `init`), you have to decide whether it have to be accepted (*Check Requirements*). This can be modelled by the rule

```
[init(P)@(T)]
  !receiveOreder(P)@T
 [~~r~~eceiveOrder(P)@T]
  !checkRequirements(P)@T++ ←
```

In the case you find the order convenient, you notify your decision (*Accept Order*) and you start with the actual realization:

```
[OK(P)@T]
  !acceptOrder(P)@T++ ←
```

Otherwise the order will be refused:

```
[notOK(P)@T]
  !refuseOrder(P)@T++ ←
```

Then, when the order is accepted you will start the actual realization and you will eventually release the tool. Here, we have assumed just two phases: *Requirement Analysis* and *Implementation*. Moreover, if some developer leave the company during the implementation this may cause the failing of the project. In any case, the tool must be released within some time bound. The above tasks can be modelled in an high-level way as previously done.

The power of $\mathtt{DATALOG}^{!ev}$ is that it allows to model in a very simple way also the subtasks. For instance, in the same figure, the *Check Requirement* activity has been described. It comprises a check for the reliability of the client, that, for instance, may take 1 day. Moreover, there is an other activity of staffing the employees to the project. The activity *Check Requirements* can be formalized as follows:

```
[checkRequirement(P)@T]
  !verifyClient(P)@T++ ←
```

where `verifyClient` is not modelled here. However, we assume that it may trigger either the event `notOK` (if the client is not reliable) or the event `verifyStaffing` which is responsible for checking also the staffing — notice that the staffing is performed in a new dimension, since this is a conceptual activity.

```
[verifyStaffing(P)@T]
   !staff(P)@sub(T) ←
```

The staffing of the project is another complex task. In fact, 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 this staffing process are not relevant. Rather, it is crucial that such a process is correctly performed, according to the above constraints.

For a given project, we want to set up a team of employees such that: (i) an employee can be assigned to one project at a time, (ii) for each skill required in the project, one employee must be present in the working team. A project is *staffed* if both the above conditions are satisfied.

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 4 time units and requires a developer and a consultant, that employee e_1 has the skill s_1 (developer), etc.

We next define a DATALOG$^{!ev}$ program $\mathcal{P}_s$ that represents our staffing problem – here we give just a flavor of its meaning, as the semantics of the language is presented in the next section. $\mathcal{P}_s$ contains a set of DDB facts `inTeam(Project#, Employes#)@t` to encode the fact that an employee is enrolled in a project at some time `t`. The following D-KB rules in $\mathcal{P}_s$ determine whether a project P is staffed at a time T.

```
staffed(P)        ←   project(P,_), ¬missingSkill(P).
missingSkill(P)   ←   requiredSkill(P,S,_),
                      ¬skillInP(S,P).
skillInP(S,P)     ←   inTeam(P,E), employee(E,S,_).
```

Moreover, $\mathcal{P}_s$ also contains the following rules for inferring the employees that are not currently involved into any project, and whose skills are still missing.

```
candidate(E,P)       ←   employee(E,S), ¬inSomeTeam(E),
                         requiredSkill(P,S,_),
                         ¬skillInP(S,P).
inSomeTeam(E)        ←   project(P,_), inTeam(P,E).
moreCandidates(P)    ←   candidate(E,P).
```

The only event transition rule for the staffing problem has been already shown in Figure 3. Roughly speaking, the `staff` event is responsible for choosing *non-deterministically* any employee for the inclusion in the project having a skill that is still missing in the staffing. After the project is staffed the event `OK` is eventually triggered. Otherwise, i.e., if there are no more candidates but the project is not yet staffed, then `notOK` is triggeered.

Finally, at the end of the project we release the team members (note that, in this case, we do not choice 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.

requiredSkill:

P#	Skill#	Description
p_1	s_1	developer
p_1	s_2	consultant
p_2	s_1	developer
p_2	s_3	researcher

project:

P#	Duration
p_1	4
p_2	3

employee:

E#	Skill#
e_1	s_1
e_1	s_1
e_2	s_2
e_3	s_3

Figure 4: An extensional database EDB for the program $\mathcal{P}$ in the running example.

```
[end(P)@T]
   −inTeam(P,E) ← 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 your 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$^{!ev}$ might help your job. In fact, after the process is modelled, given some orders you might be interested in verifying whether there exists a proper staffing that guarantees the satisfaction of all the order (and, obviously, in getting such a *plan*). Moreover, you might be interested in identifying the employees which are crucial for the implementation of a tool, i.e., the employees that will cause a failure of the project after leaving the company.

5. SEMANTICS

Let $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB}} = \langle \mathtt{D\text{-}KB}, \mathtt{EV}\rangle$ be a DATALOG$^{!ev}$ program, over a time domain T and a knowledge base EDB. As usual, the *Herbrand Universe* $U_{\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB}}}$ of a $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB}}$ is the set of all constants appearing in $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB}}$.

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

Definition 5.1 An *interpretation* for the program $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB}}$ consists of a pair $\langle S, E\rangle$, where S is a set of ground literals and E is a set of ground events, such that

$$S \subseteq \mathbf{B}_{\mathtt{EDB}} \cup \mathbf{B}_{\mathtt{IDB}} \cup \mathbf{B}_{\mathtt{EV}} \cup \mathbf{B}_{\mathtt{DDB}}$$
$$E \subseteq \mathbf{B}^{+}_{\mathtt{EV}}$$

The minimum temporal argument occurring in the events in E is denoted by $nextTime(I)$, while the maximum temporal argument occurring in the predicates in S is denoted by $curTime(I)$. Finally, an interpretation I is *feasible* if $E = \emptyset$ or $curTime(I) \prec nextTime(I)$. □

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

Note that in the above definition, we assume that any DDB predicate asserted at a given time, remains valid till it is explicitly retracted from the database; indeed, the behavior of the DDB predicates is essentially *inertial*, while the truth value of the 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 *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,

$$\begin{aligned} I_1 &= \langle \{\mathtt{staffable}(\mathtt{p_1}, \mathtt{e_1})@3,\ \mathtt{inTeam}(\mathtt{p_2}, \mathtt{e_3})@5\}, \\ &\quad \{\mathtt{end}(\mathtt{p_1})@8\}\rangle \end{aligned}$$

and

$$\begin{aligned} I_2 &= \langle \{\mathtt{inTeam}(\mathtt{p_2}, \mathtt{e_3})@8, \neg\mathtt{inTeam}(\mathtt{p_2}, \mathtt{e_4})@5\}, \\ &\quad \{\mathtt{end}(\mathtt{p_2})@2, \mathtt{end}(\mathtt{p_1})@9\}\rangle \end{aligned}$$

are both interpretations. However, the latter is not feasible since $\mathit{nextTime}(I_2) = 2$ and $\mathit{curTime}(I_2) = 8$. Moreover, note that in the former interpretation the predicate $\mathtt{inTeam}(\mathtt{p_2}, \mathtt{e_3})$ 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 $\mathit{triggered}(I)$ the set of all events in E having temporal argument $\mathit{nextTime}(I)$, in the case $\mathit{nextTime}(I) \in \mathtt{T}$; otherwise, we let $\mathit{triggered}(I) = \emptyset$. Let $TR(I)$ be the set of all transition rules such that all their activating events belong to $\mathit{triggered}(I)$, and $C(I)$ be the set of all choice predicates occurring in the rules in $TR(I)$. Moreover, let $\mathit{ground\_TR}(I)$ be the set of all the ground instantiations R of the rules in $TR(I)$ 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, $\mathit{ground\_TR}(I)$ contains a set of enabled ground rules (coming from instantiations of the rules in $TR(I)$) for each possible way of enforcing the functional dependencies determined by the choices in $C(I)$.

Let *chosen_tr* be any set of ground rules in $\mathit{ground\_TR}(I)$. We denote by $\mathcal{A}_I(\mathit{chosen\_tr})$ the set of all the dynamic atoms p such that $+$p occurs in the head of some transition rule in *chosen_tr* and p is false w.r.t. I. Such a dynamic atom p is said to be asserted. Similarly, $\mathcal{R}_I(\mathit{chosen\_tr})$ is the set of all the dynamic literals $\neg$p such that $-$p occurs in the head of some transition rule in *chosen_tr* and p is true w.r.t. I. In this case, we say that p has been retracted. Finally, $\mathcal{E}_I(\mathit{chosen\_tr})$ is the set of the events triggered by all transition rules r in *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 $\mathcal{I}_\mathcal{P}$, while the set of all the subsets of $\mathcal{I}_\mathcal{P}$ is denoted by $2^{\mathcal{I}_\mathcal{P}}$.

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

$$\begin{aligned} S' &\in \mathrm{SM}(D\text{-}KB \cup S \cup \mathcal{A}_I(\mathit{chosen\_tr})\mathcal{R}_I(\mathit{chosen\_tr})) \cup \\ &\quad \cup \mathit{triggered}(E), \text{ and} \\ E' &= E \cup \mathcal{E}_I(\mathit{chosen\_tr}) - \mathit{triggered}(E). \end{aligned}$$

□

Note that, for any given interpretation $I = \langle S, E\rangle$, this function computes the set of all feasible 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 $\mathit{nextTime}(I)$. All these events are removed from the set of envisioned events E', while new events possibly planned to occur in the future are added to E' through the set $\mathcal{E}_I(\mathit{chosen\_tr})$. 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 choices constructs, the output of $\mathbf{T}$ applied on a singleton $\{I\}$ is in general a set of multiple alternative interpretations, even in the case the dynamic knowledge base is stratified ($\mathtt{DATALOG}^{!\mathtt{ev},\neg_\mathtt{s}}$ 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 $\mathtt{DATALOG}^{!\mathtt{ev}}$ 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 $\mathcal{P}_{\mathtt{EDB},\mathtt{H}}$) is the succession of sets of interpretations $\widehat{\mathbf{T}}$ such that (i) $\widehat{\mathbf{T}}_\mathbf{0} = \{\langle \mathtt{EDB}, \mathtt{H}\rangle\}$, and (ii) $\widehat{\mathbf{T}}_{\mathtt{i+1}} = \mathbf{T}(\widehat{\mathbf{T}}_\mathtt{i})$.

For every $j > 0$, any interpretation $M \in \widehat{\mathbf{T}}_j$ is called a *temporal model* for $\mathcal{P}_{\mathtt{EDB},\mathtt{H}}$. □

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 $\mathtt{DATALOG}^{!\mathtt{ev}}$. For instance, in our running example the list $[\mathtt{init}(\mathtt{p_1})@0, \mathtt{init}(\mathtt{p_2})@3]$ is used for simulating a scenario in which two projects are going to be staffed at times 0 and 3, respectively. Under an abstract perspective, the events in H are used for constraining the evolution of the $\mathtt{DATALOG}^{!\mathtt{ev}}$ program.

Definition 5.5 Let $\mathcal{P}$ be a $\mathtt{DATALOG}^{!\mathtt{ev}}$ program, EDB be an input database, and H a list of ground events. A temporal model M for $\mathcal{P}_{\mathtt{EDB},\mathtt{H}}$ is a *stationary model* (for $\mathcal{P}_{\mathtt{EDB},\mathtt{H}}$) if it is a fixpoint of $\mathbf{T}$, i.e., if $M \in \mathbf{T}(\{M\})$. Then, $\mathit{curTime}(M)$ 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 $\mathcal{P}_{\mathtt{EDB},\mathtt{H}}$ is stationary if and only if $E = \emptyset$.*

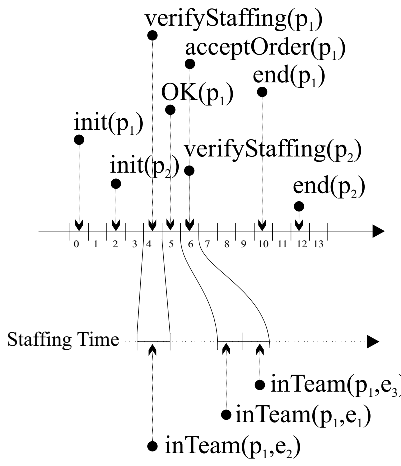


Figure 5: A graphical view of a temporal model for $\mathcal{P}_s$.

Example 5.7 Assume that the staffing of project p_1 should start at the time instant 0 and the staffing of project p_2 at the time instant 3. This is encoded through the list of envisioned events $\mathtt{H} = [\mathtt{init}(\mathtt{p}_1)@0, \mathtt{init}(\mathtt{p}_2)@2]$. Then, the graphic reported in Figure 5 shows (the relevant atoms of) a temporal model of $\mathcal{P}_s$, 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 $\langle 6, 1\rangle$ and $\langle 6, 2\rangle$) 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 $\mathcal{P}$ be a* DATALOG$^{!ev}$ *program,* EDB *be an input database,* H *a list of ground events, and I be an interpretation of $\mathcal{P}$. 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 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 $\mathcal{P}_{\mathtt{EDB,H}}$ is denoted by $\mathcal{TM}(\mathcal{P}_{\mathtt{EDB,H}})$ (resp. $\mathcal{TSM}(\mathcal{P}_{\mathtt{EDB,H}})$).

6. QUERIES AND COMPLEXITY ISSUES

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

DATALOG$^{!ev}$ *queries* are formulae 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. to the $\preceq$ ordering). Then, $\forall^{@\mathtt{t}}\mathtt{C}$ and $\exists^{@\mathtt{t}}\mathtt{C}$ are queries. Moreover, let $\mathtt{t}_1$ and $\mathtt{t}_2$ be two time instants such that $\mathtt{t}_1 \prec \mathtt{t}_2$, let C be a (possibly empty) conjunction of ground literals with time arguments at most $\mathtt{t}_1$, and let Q be a query, whose first quantifier is either $\exists^{@\mathtt{t}_2}$ or $\forall^{@\mathtt{t}_2}$. Then, $\forall^{@\mathtt{t}_1}\mathtt{C}\wedge\mathtt{Q}$ and $\exists^{@\mathtt{t}_1}\mathtt{C}\wedge\mathtt{Q}$ 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 DATALOG$^{!ev}$ program, and a time t, we denote by $M@t = \langle S', E\rangle$ the interpretation consisting of all the atoms having any temporal argument $t' \preceq t$.

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

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

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

Otherwise, we say that Q is false w.r.t. $\mathcal{M}$.

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

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

Example (contd.) Consider again the staffing program $\mathcal{P}_s$ in our running example, the extensional database of Figure 4, and the list of envisioned events $\mathtt{H} = [\mathtt{init}(\mathtt{p}_1)@0, \mathtt{init}(\mathtt{p}_2)@3]$.

Then, consider the query $Q_1 : \exists^{@6}(\mathtt{end}(\mathtt{p}_1) \wedge \exists^{@7}\mathtt{end}(\mathtt{p}_2))$. 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 6, and there is an evolution of this model where p_2 is completed within time 7. In our case, Q_1 is in fact true, as witnessed by the temporal model in 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.

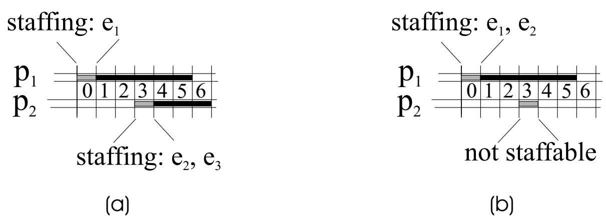


Figure 6: Two evolutions for the program $\mathcal{P}_s$.

Now, consider the (more stringent) query $\forall^{@6}(\mathtt{end(p_1)} \wedge \exists^{@7}\mathtt{end(p_2)})$. It is easy to verify that this query evaluates to false for the program $\mathcal{P}_s$, given EDB and H. Indeed, for instance, the temporal model in Figure 6.b is an evolution (from time 6) of a temporal model such that p_1 ends at time 6; however, in this model p_2 cannot be executed at all. Indeed, p_2 requires a developer, but both e_1 and e_2 are already busy for they are participating to p_1. □

6.1 Computational Complexity: setting and assumptions

We next study the computational complexity of a number of problems related to $\mathtt{DATALOG}^{!ev}$ programs and queries. In particular, following the *data complexity* approach [34], in all results stated below we will consider a given problem instance having as its input the temporal domain T, the extensional database EDB, and the list of envisioned events H, while both the program $\mathcal{P}$ and (possibly) the query Q are fixed.

Moreover, since the general problem is (quite trivially) undecidable, we next focus on the problem instances where the time domain T is $T^{\omega_1,m}$, with an unbounded main dimension and a number of bounded range auxiliary dimensions (see Section 3), and $\sigma^{vars} \cap \sigma^{time\_vars} = \emptyset$, i.e., variables and time variables occurring in $\mathcal{P}$ are taken from disjoint sets.

First, we discuss the problem of deciding the existence of temporal and stationary temporal models. For space reasons, we omit theorems proofs.

Theorem 6.2 (Stationary model existence) *Deciding whether $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB,H}}$ has a stationary model is* PSPACE*-complete. Hardness holds even if $\mathcal{P}$ is stratified.*

Note that the problem is much more easier, if we are satisfied with any temporal model, rather than requiring that the model is stationary.

Theorem 6.3 (Temporal model existence) *Deciding whether $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB,H}}$ has a temporal model is* NP*-complete. However, if $\mathcal{P}$ is stratified, then $\mathcal{P}^{\mathtt{T}}_{\mathtt{EDB,H}}$ always have temporal models. Moreover, any temporal model can be computed in polynomial time.*

It turns out that stratified programs have an efficient implementation as far as as the computation of one temporal model is concerned. However, if we have to answer a given query Q, and hence we are interested in some "particular" temporal models, then the complexity becomes much higher. Actually, it turns out that the complexity of answering such temporal queries spans the polynomial hierarchy, as stated by the following theorem.

Theorem 6.4 (Query answering under temporal models) *If the query Q is k-existential (resp., k-universal), deciding whether the answer $Q(\mathcal{P}^{T}_{\mathtt{EDB,H}})$ is true is Σ^P_k-complete (resp. Π^P_k-complete). Hardness holds even if $\mathcal{P}$ is stratified.*

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

Theorem 6.5 (Query answering under stationary models) *Deciding whether the answer $Q(\mathcal{P}^{T}_{\mathtt{EDB,H}})$ is stationarily true is* PSPACE*-complete.*

7. RELATED WORK AND CONCLUSION

In this paper we have presented an extension of DATALOG with events and choice, called $\mathtt{DATALOG}^{!ev}$, 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.

A distinguished feature of the 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 [28] extended the single timeline of EC into a totally ordered set of different timelines $\{T_1, ..., T_n\}$, such that each T_i is of a finer granularity that T_{i-1}. Similar ideas have been applied in [11], which also considers indeterminacy in the occurrence of events (for this latter aspect see also [17, 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 modelling 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: $\mathtt{DATALOG}^{!ev}$ 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. Rather to event calculus, our language is closer to Dynamic Logic Programming, which extends logic programming with amenities for modelling 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 [24]. 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 a certain condition 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 [16]. 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 [2] proposed EVOLP, in which we specify some rules updating the original program. Each time the 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 $\texttt{DATALOG}^{!ev}$ we stress two important differences: (i) The paradigm of multidimensional updates cannot easily fit the need of representing multiple time dimension; in fact, the above mentioned works assume only one temporal dimensions 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, $\texttt{DATALOG}^{!ev}$ has been specifically designed for being queried in order to perform temporal reasoning.

When comparing $\texttt{DATALOG}^{!ev}$ 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 $\mathcal{A}$, $\mathcal{B}$ (see, for a comparison, [19]) and $\mathcal{AC}$ [32]. It also provides a set of primitives for reasoning on past events, thus capturing the power of *Past Temporal* $\mathcal{A}$. Moreover, it deals with concurrent actions such as languages $\mathcal{C}$ [20] and $\mathcal{A}_C$ [6], and it enables to reason about actual and hypothetical occurrences of concurrent and non-deterministic actions, such as language $\mathcal{L}_2$ [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 calculus, are close in their spirit as they aims at modelling 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 effect of the actions in themselves.

Depending on the user needs, $\texttt{DATALOG}^{!ev}$ 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 in order to achieve a given temporal goal. Hence, the language shares the same perspective of [5].

We conclude by pointing out that $\texttt{DATALOG}^{!ev}$ is essentially an extension of DATALOG and hence should be compared to the different proposals of extending logic programming with temporal logics (see, for a survey of the different proposals, [30]). Among the first proposals, we recall *Datalog*$_{1S}$ [12] and *Templog* [1]. The former is DATALOG extended with one successor modelling 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*$_{1S}$ seems a rather trivial extension of DATALOG) it has been proved that they are equivalent as for the expressiveness and the completeness. Starlog [13] is another logic language that adds an additional time-argument to every PROLOG predicate, and that is designed for general purpose programming, for simulation, and for modelling 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, these are very unpractical for modelling complex situations in which a knowledge base is updated due to the occurrence of actions.

Finally, it is worth mentioning that we are implementing $\texttt{DATALOG}^{!ev}$ as a front end extending the disjunctive Datalog system `dlv` [25]. This implementation is based on an extension of the notion of XY-stratification, which has been used to model updated and active rules [35].

8. REFERENCES

[1] M. Abadi and Z. Manna. Temporal Logic Programming. *JSC*, 8(3), 1989.

[2] J. J. Alferes, A. Brogi, J. A. Leite, and L.M. Pereira. Evolving Logic Programs. In *Proc. of JELIA'02*, pages 50–61, 2002.

[3] J. J. Alferes, J. A. Leite, L. M. Pereira,H. Przymusinska, and T. C. Przymusinski. Dynamic Logic Programming. In *Proc. of KR'98*, pages 98–111, 1998.

[4] J. J. Alferes, L. M. Pereira,H. Przymusinska, and T. C. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 138(1–2):87–116, 2002.

[5] C. Baral. A Systematic Approach to reason with time and situations. *Unpublished.*

[6] C. Baral, and M. Gelfond. Reasoning About Effects of Concurrent Actions. *JLP*, 31(1-3):85–117, 1997.

[7] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypotheses. *JLP*, 31(1-3):201–243, 1997.

[8] C. Bettini, X. Sean Wang, and S. Jajodia. A General Framework for Time Granularity and Its Application to Temporal Reasoning. *AMAI*, 22(1-2):29–58, 1998.

[9] I. Cervesato, M. Franceschet, and A. Montanari. A Guided Tour through Some Extensions of the Event Calculus. *Computational Intelligence*, 16(2):307–347, 2000.

[10] I. Cervesato, and A. Montanari. A General Modal Framework for the Event Calculus and Its Skeptical and Credulous Variants. *JLP*, 38(2):111–164, 1999.

[11] L. Chittaro, and C. Combi. Temporal Granularity and Indeterminacy in Reasoning About Actions and Change: An Approach Based on the Event Calculus. *Annals of Mathematics and Artificial Intelligence*, 36(1–2):81–119, 2002.

[12] J. Chomicki, and T. Imielinski. Relational Specifications of Infinite Query Answers. In *Proc. of SIGMOD'89*, pages 174–183, 1989.

[13] R. Clayton, J. Cleary, B. Pfahringer, and M. Utting. Starlog homepage. `http://www.scms.waikato.ac.nz/cs/Research/starlog/`.

[14] Curtis E. Dyreson, W. S. Evans, Hong Lin, and R. T. Snodgrass. Efficiently Supported Temporal Granularities. *TKDE*, 12(4):568–587, 2000.

[15] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *TOCL*, 2003. To appear.

[16] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A Framework for Declarative Update Specifications in Logic Programs. In *Proc. of IJCAI'01*, pages 649–654, 2001.

[17] M. Franceschet, and A. Montanari. A graph-theoretic approach to efficiently reason about partially ordered events in (Modal) Event Calculus. *AMAI*, 30(1-4):93–118, 2000.

[18] M. Gelfond, and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP/SLP'88*, pages 1070–1080, 1988. MIT Press.

[19] M. Gelfond, and V. Lifschitz. Action Languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.

[20] E. Giunchiglia, and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proc. of AAAI '98*, pages 623–630, 1998.

[21] I. A. Goralwalla, Y. Leontiev, M. T. Özsu, D. Szafron, and C. Combi. Temporal Granularity: Completing the Puzzle. *Journal of Intelligent Information Systems*, 16(1):41–63, 2001.

[22] S. Hanks, and D. McDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[23] R. A. Kowalski, and M. J. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.

[24] J. A. Leite, J. J. Alferes, and L. M. Pereira. Multidimensional Dynamic Knowledge representation. In *Proc. of LPNMR'01*, pages 365–378, 2001. Lecture Notes in AI (LNAI).

[25] N. Leone, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, and A. Polleres. The DLV System. In *Proc. of JELIA'02*, LNCS 2424, pages 537–540, 2002.

[26] V. W. Marek, and Mirosław Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[27] J. McCarthy, and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[28] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with time granularity in the Event Calculus. In *Proc. of FGCS'92.* , pages 702–712, 1992.

[29] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Mirosław Truszczyński, editors, *Proc. of NMR'00*, 2000.

[30] M. Ali Orgun, and Wanli Ma. An Overview of Temporal and Modal Logic Programming. In *Proc. of ICTL'94*, pages 445–479, 1994. Springer-Verlag.

[31] G. J. Sussman. The Virtuous Nature of Bugs. *Readings in Planning*, chapter 3, pages 111–117. Morgan Kaufmann, 1990.

[32] H. Turner. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming*, 31(1–3):245–298, 1997.

[33] H. Turner. A Logic of Universal Causation. *Artificial Intelligence*, 113:87–123, 1999.

[34] M. Vardi. The Complexity of Relational Query Languages. In *Proc. of STOC'82*, pages 137–146, 1982.

[35] C. Zaniolo. Active Database Rules with Transaction-Conscious Stable Model Semantics. In *Proc. of DOOD'95*, pages 55–72, LNCS 1013,1995.