What the Event Calculus actually does, and how to do it efficiently

Iliano Cervesato*, **Luca Chittaro**+, **Angelo Montanari**+

(*) Dipartimento di Informatica
Università di Torino
Corso Svizzera, 185
10149 Torino - ITALY
iliano@di.unito.it

(+) Dipartimento di Matematica e Informatica
Università di Udine
Via Zanon, 6
33100 Udine - ITALY
{chittaro | montana}@dimi.uniud.it

Abstract

Kowalski and Sergot's Event Calculus (EC) is a formalism for reasoning about time and change in a logic programming framework. From a description of events which occur in the real world and properties they initiate and terminate, EC allows to derive the maximal time intervals (MVIs) over which properties hold. In this paper, we provide a precise characterization of what EC actually does to compute MVIs, and propose a solution to do it efficiently when incomplete information about event ordering is given. We first introduce the basic features of EC with relative times and partial ordering, and show that it requires an exponential time to compute MVIs for a given property. Then, we show how adopting graph marking techniques reduces the exponential complexity to a polynomial cost. Successively, we formally introduce the notion of kernel of an ordering relation, and show that it can be usefully applied to further reduce the complexity of computing MVIs.

1 Introduction

Kowalski and Sergot's Event Calculus (EC) is a formalism for representing and reasoning about events and their effects in a logic programming framework [3, 7, 9, 10, 11]. Given a set of *events* occurring in the real world, EC is able to infer the set of maximal validity intervals (MVIs, hereinafter) over which the *properties* initiated and/or terminated by the events maximally hold. Event occurrences can be provided with different temporal qualifications [1]. In this paper, we suppose that for each event we either specify its relative position with respect to some other events (e.g., event e_1 occurs before event e_2) or leave it temporally unqualified (the only thing we know is that it occurred). Depending on temporal qualifications, the sequence of events can be totally or partially ordered. Different variants of EC can be obtained by varying the type of temporal qualifications and the amount of ordering information. These variants range from the strict version with absolute times and total ordering [4] to the loose one with only relative times and partial ordering [5]. It is possible to show that reducing the information about event ordering increases the complexity of computing all MVIs for a given property p. In EC with absolute times and total ordering, the worst-case complexity is indeed $O(n^3)$, where n is the number of recorded events initiating or terminating p, while it becomes $O(2^n)$ in the loose setting.

In this paper, we provide a precise characterization of what EC actually does to compute MVIs, and propose a solution to do it efficiently when incomplete information about event ordering is given. We consider an EC database consisting of a set of events $E=\{e_1, ..., e_n\}$ and a set w of elements (e_i,e_j) whose transitive closure w^+ is a strict ordering relation on $E\times E$. The set w can be modeled as a directed acyclic graph G, whose nodes are event occurrences, such that there exists an edge from node e_i to e_j if and only if the pair (e_i,e_j) belongs to w. Database updates in EC provide information about the occurrences of events and their times [7]. In this paper, we assume that the set of events is fixed, and the input process consists in the addition of ordering information. EC updates are of additive nature only, and their cost is constant. In order to compute the set of MVIs for a given property p, for each pair of events e_i, e_j such that e_i initiates p and e_j terminates p, EC first checks whether (e_i,e_j) belongs to w^+ and then it ascertains that no event interrupting p occurs between e_i and e_j. Such an approach presents two major drawbacks. First of all, EC blindly picks up every pair consisting of an event initiating p and an event terminating p, and only later looks for possible interruptions. Moreover, in the case of incomplete ordering information, EC searches for a path from e_i to e_j in G in order to check whether the ordered pair of events (e_i,e_j) belongs to w^+. If no memory of the visited nodes is kept, such a search can involve an exponential number of lookups to the nodes of G. To overcome these limitations, we first introduce a marking technique that speeds up the visit of G, reducing the order of complexity to $O(n^5)$. Then, we modify the management of updates in order to restrict the selection of the pairs of events respectively initiating or terminating a given property to those pairs for which the existence of an interrupting event can be a priori excluded. This is done by taking into account the least subset w^- of w such that $w^+ = (w^-)^+$ (w^- is called the *kernel* of w). We revise update processing in order to record and maintain w^- instead of w. This makes it possible to constrain the computation of MVIs to search for the ordered pairs belonging to w^- only. As suggested by experimental results, we expect this revision to result into a further reduction in complexity. At the moment, we have proved that in the case where all events in the database affect the same property, we can reach a complexity as low as $O(n^2)$.

The paper is organized as follows. We first introduce the basic features of EC with relative times and partial ordering, and analyze its computational complexity. Then, we show how the addition of marking reduces the exponential complexity of the calculus to a polynomial cost. Successively, we formally introduce the notion of kernel of an ordering relation, and prove that it can be usefully applied to further reduce the complexity of computing MVIs.

2 The Event Calculus with relative times and partial ordering

EC proposes a general approach to representing and reasoning about events and their effects in a logic programming framework. It takes the notions of event, property, time-point and time-interval as primitives and defines a model of change in which *events* happen at *time-points* and initiate and/or terminate *time-intervals* over which some *property* holds. Time-points are unique points in time at which events take place instantaneously. Time-intervals are represented as pairs of time-points. EC embodies a notion of *default persistence* according to which properties are assumed to persist until an event occurs which terminates them. A specific domain evolution is called a *history* and is modeled by a set of event occurrences. The calculus allows to infer the set of time intervals over which the properties initiated and/or terminated by event occurrences maximally hold, i.e. the MVIs. Since it has been defined in the framework of logic programming, EC can derive MVIs by running its axiomatic definition as a Prolog program.

In this paper, we focus our attention on situations where precise temporal information for event occurrences is not available. We represent the occurrence of an event *e* by means of the clause:

```
happens(e)
```

The relation between events and properties is defined by means of **`initiates`** and **`terminates`** clauses:

```
initiates(e, p).                          terminates(e, p).
```

The **`initiates`** (**`terminates`**) clauses relate each event *e* to the property *p* it initiates (respectively terminates). We assume that each event initiates/terminates a period of validity for a single property.

The plain EC model of time and change is defined by means of the axioms:

```
holds(period(Ei,P,Et)):-           (1.1)
   happens(Ei), initiates(Ei,P),
   happens(Et), terminates(Et,P),
   before(Ei,Et),
   not broken(Ei,P,Et).
```

```
broken(Ei,P,Et):-                               (1.2)
   happens(E),
   before(Ei,E), before(E,Et),
   (initiates(E,Q);terminates(E,Q)),
   exclusive(P,Q).
```

The first axiom states that a property *P* maximally holds between events *Ei* and *Et* if *Ei* initiates *P* and occurs before *Et* that terminates *P*, provided there is no known interruption in between. The negation involving the predicate **`broken`** is interpreted using negation-as-failure. The second axiom states that a given property *P* does not hold uninterruptedly between *Ei* and *Et* if there is an event *E* that happens between them and initiates or terminates a property *Q* that is incompatible with *P*. The predicate **exclusive(***P***,** *Q***)** has been introduced as a constraint to force the derivation of *P* to fail when it is possible to conclude that *Q* holds at the same time, and vice versa [8]. Furthermore, by adding the axiom:

```
exclusive(P, P).                   (1.3)
```

we guarantee that **`broken`** succeeds also when an initiating or terminating event for property *P* is found between the pair of events *Ei* and *Et* starting and terminating *P*, respectively.

Finally, knowledge about the relative ordering of events is expressed by means of facts of the form **beforeFact(**e_1**,** e_2**)**. The predicate **`before`** used in **`holds`** and **`broken`** is defined as the transitive closure of **`beforeFact`**:

```
before(E1,E2):-                    (1.4)
   beforeFact(E1,E2).
```

```
before(E1,E2):-                    (1.5)
   beforeFact(E1,E3),
   before(E3,E2).
```

The ordering information is entered through the predicate **updateOrder(**e_1**,** e_2**)**:

```
updateOrder(E1, E2) :-             (1.6)
   assert(beforeFact(E1, E2)).
```

We assume that the set of ordered pairs is always consistent as it grows. This means that **`before`** is supposed to represent a relation that is irreflexive, anti-symmetric and transitive.

The axioms of EC, shown as clauses (1.1-6), will be referred as program 1 in the following.

3 A complexity analysis

In the case of EC with absolute times and total ordering, the worst case complexity of deriving all the MVIs for a given property has been proven to be $O(n^3)$, where n is the number of recorded events for a given property [4]. In this section, a worst case complexity analysis will be carried out for EC with relative times and partial ordering. The cost is measured as the number of accesses to the database to unify facts during the computation. Some hashing mechanism is assumed so that fully instantiated atomic goals are matched in one single access to a sequence of variable-free facts in case of success, and do not need any access in case of failure. The complexity is given as a function of the number n of recorded events.

3.1 The complexity of EC with relative times and partial ordering

Queries have the form **`holds(period(Ei,p,Et))`**, where **`Ei`** and **`Et`** are variables and **`p`** can be either a variable or a constant. The update predicate is always called with ground arguments. For each predicate, we now analyze the cost of finding all its solutions.

- **`updateOrder(e1,e2)`**: a call to this predicate has unitary cost since it only results in asserting a new fact in the database.
- **`happens(E)`**: this goal succeeds n times, since n events are recorded in the database. So, its cost is $O(n)$.
- **`initiates(e,p)`** and **`terminates(e,p)`**: the cost of these predicates is constant for a ground **`e`** even when called with **`p`** uninstantiated (as in clause 1.2). Indeed, they are functional in their second argument since an event determines univocally its associated property. It remains constant also when an event is allowed to initiate and terminate more than one property.
- **`exclusive(p,q)`**: this predicate is always called ground. It can be matched against at most one fact in the database (possibly clause (1.3)). The cost is therefore constant.
- **`beforeFact(e1,e2)`**: When called ground, as in clause (1.4), the query cost is constant. In clause (1.5) instead, the call results in instantiating a variable. The complexity is given as the maximum number of matching facts in the database. Having n nodes, at most n-1 edges can start from a given node. Thus, when called with one variable argument, this predicate has cost $O(n)$.
- **`before(e1,e2)`**: we will show that the standard two-clauses definition of **`before`** (clauses 1.4-5) has a worst case complexity that is at least exponential. Consider $n \geq 4$ events arranged as shown in figure 1: all the n events but two ($\mathbf{e_{n-1}}$ and $\mathbf{e_n}$) participate in a total order between $\mathbf{e_1}$ and $\mathbf{e_{n-2}}$. All the transitive pairs, but the pair

(e_1,e_n) are explicitly specified by means of **beforeFact**. We assume that **beforeFact(e_1,e_{n-1})** textually follows any other fact of the form **beforeFact(e_1,e_i)**, for **i** = **2**..**n-2**.

Due to the operational behavior of Prolog, EC thus tries to prove **before(e_1,e_n)** looking for a path that passes through **e_{n-2}** before attempting the (only possible) path that includes **e_{n-1}**. Backtracking due to the failure in proving **before(e_{n-2},e_n)** causes every path from **e_1** to **e_{n-2}** to be unsuccessfully attempted. The resulting cost is computed as follows. Let $m = n$-3 be the length of the longest path between **e_1** and **e_{n-2}**. We have the following recursive relation, where $C_{\mathbf{before}}(m)$ represents the cost of finding all the solutions to the goal **before(e',e")** in case a total order of length m exists between **e'** and **e"** (e.g. the value of m highlighted in Figure 1 concerns the pair **(e_1,e_{n-2})**):

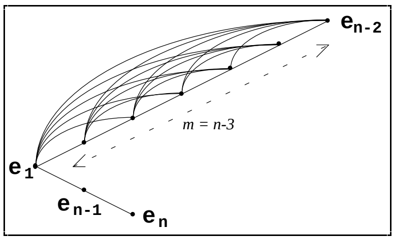


Figure 1

$$\begin{cases} C_{\mathbf{before}}(1) = 1; \\ C_{\mathbf{before}}(m) = m + \sum_{i=1}^{m-1} C_{\mathbf{before}}(i) \qquad m > 1 \end{cases}$$

Indeed, there are m edges (represented by **beforeFact**) starting from **e'**, which we can traverse to go towards **e"**. One of these edges directly reaches **e"** (clause 1.4). If $m > 1$, each of the remaining m-1 edges leads to a node **e** (clause 1.5), reducing the problem to size i, where i is the length of the longest path between **e** and **e"**.

In order to give an analytical form for $C_{\mathbf{before}}(m)$, let us unfold the expression for $C_{\mathbf{before}}(m)$:

$$\begin{aligned} C_{\mathbf{before}}(m) &= m + C_{\mathbf{before}}(m\text{-}1) + C_{\mathbf{before}}(m\text{-}2) + C_{\mathbf{before}}(m\text{-}3) + \dots + C_{\mathbf{before}}(1) \\ &= m + (m\text{-}1) + 2C_{\mathbf{before}}(m\text{-}2) + 2C_{\mathbf{before}}(m\text{-}3) + \dots + 2C_{\mathbf{before}}(1) \\ &= m + (m\text{-}1) + 2(m\text{-}2) + 4C_{\mathbf{before}}(m\text{-}3) + \dots + 4C_{\mathbf{before}}(1) \\ &= \quad \dots \\ &= m + 2^0(m\text{-}1) + 2^1(m\text{-}2) + 2^2(m\text{-}3) + \dots + 2^{m\text{-}2}\cdot 1 \end{aligned}$$

We can summarize this formula as:

$$C_{\mathbf{before}}(m) = m + \sum_{i=1}^{m-1} 2^{i-1}(m-i) \geq \sum_{i=1}^{m-1} 2^{i-1} = \sum_{j=0}^{m-2} 2^j = 2^{m-1} - 1$$

Therefore the cost is at least $O(2^m)$. Since we set $m = n$-3, **before** with both arguments instantiated turns out to be at least exponential in the worst case.

- **broken(ei,p,et)**: since the definition of this predicate contains calls to **before**, this goal has an exponential complexity.
- **holds(period(Ei,p,Et))**: reasoning as for **broken**, we obtain an exponential cost too.

The major results of the preceding analysis are the constant cost of updating ordering information (**`updateOrder`**) and the exponential query complexity (**`holds`**).

3.2 The addition of marking

The exponential cost resulting from the previous analysis makes EC with relative timing not appealing for practical computations. It is interesting to note that this very high cost origins from the calls to **`before`**. Can this predicate be re-implemented with a lower cost?

We use **`before`** to check whether a pair of nodes belongs to the transitive closure of a cycle-free relation. Well-known algorithms for this kind of operations have polynomial complexity in the number of nodes. Unfortunately, these algorithms are more suited to be implemented using traditional programming languages rather than logic programming.

We will now present a Prolog program implementing a marking algorithm. We foresay that it is written using extra-logical features of Prolog. Therefore, it will hardly be classified as a logic program. Nevertheless, this program can be considered acceptable: once we have proven that the two versions of **`before`** behave coherently, we can see the non-declarative procedure as an actual implementation of the logical one.

The idea is very simple: during the search, nodes are marked as they are visited, and only edges leading to not yet marked nodes are analyzed. We need to change the arity of the predicate **`happens`** to support marking: the second argument contains either **`marked`** or **`unmarked`** with the obvious meaning. Initially all the nodes are unmarked.

```
before(E1, E2) :-                      (2.1)
   markingBefore(E1, E2), !,
   unmarkAll.
before(E1, E2) :-                      (2.2)
   unmarkAll,
   fail.

markingBefore(E1, E2) :-               (2.3)
   beforeFact(E1, E2), !.
markingBefore(E1, E2) :-               (2.4)
   beforeFact(E1, E3),
   happens(E3, unmarked),
   mark(E3),
   markingBefore(E3, E2).
```

```
unmarkAll :-                           (2.5)
   happens(E, marked), !,
   unmark(E),
   unmarkAll.
unmarkAll.                             (2.6)

unmark(E) :-                           (2.7)
   retract(happens(E, marked)),
   assert(happens(E, unmarked)).

mark(E) :-                             (2.8)
   retract(happens(E, unmarked)),
   assert(happens(E, marked)).

happens(E) :-                          (2.9)
   happens(E, _).
```

Program 2

Clause (2.9) serves to maintain the one parameter interface to **`happens`**.

Let us now prove that the two versions of **`before`** compute the same relation, given the same factual database. The declarative program consisting of clauses (1.4-5) yields $\mathtt{before}(\mathbf{e}_1,\mathbf{e}_f)$ if and only if the database contains a path leading from $\mathbf{e}_1$ to $\mathbf{e}_f$, where the edges are represented by **`beforeFact`** (a simple inductive proof can be found in [2]). We will now prove that the database contains that path if and only if program 2 derives $\mathtt{markingBefore}(\mathbf{e}_1,\mathbf{e}_f)$, and consequently $\mathtt{before}(\mathbf{e}_1,\mathbf{e}_f)$.

First suppose that the database contains a path $p = (\mathbf{e}_1,\mathbf{e}_2),(\mathbf{e}_2,\mathbf{e}_3),\ldots,(\mathbf{e}_{k\text{-}1},\mathbf{e}_k),(\mathbf{e}_k,\mathbf{e}_f)$ of length k between nodes $\mathbf{e}_1$ and $\mathbf{e}_f$ and that all nodes are initially unmarked (the validity of this condition after each execution of **`before`** is guaranteed by the execution of the **unmarkAll** predicate occuring in clauses (2.1) and (2.2). Let us prove that **markingBefore(e_1,e_f)** is derivable from program 2, and only nodes **e_2**,**e_3**,..,**e_k** of the selected path are marked during this process. The proof is inductive in the length *n* of this path. The case *n*=1, i.e. *p* = (**e_1**,**e_f**), is caught by clause (2.3). We assume, as inductive hypothesis, that the statement holds for length *n*-1, and we prove that it holds for length *n*. Let us consider an instance of clause (2.4) where **E1** = **e_1** and **E2** = **e_f**. The first subgoal succeeds for the presence of **beforeFact(e_1,e_2)** in the database, instantiating **E3** to **e_2**. By hypothesis, all nodes are initially unmarked and thus the second subgoal, **happens(e_2,unmarked)**, is immediately provable. Moreover, the presence of this fact in the database causes the subgoal **mark(e_2)** to mark **e_2** by clause (2.8). By inductive hypothesis, **markingBefore(e_2,e_f)** is derivable and nodes **e_3**,..,**e_n** of the selected path are marked as a by-product of the process. Thus, the overall clause succeeds and proves the goal **markingBefore(e_1,e_f)**.

Conversely, suppose that **markingBefore(e_1,e_f)** is derivable from program 2. We proceed by induction on the height *h* of the resolution tree for **markingBefore(e_1,e_f)**. If *h* = 1, then clause (2.3) has been applied, and the database contains the fact **beforeFact(e_1,e_f)**. Otherwise, we assume, as inductive hypothesis, that the statement holds for every tree of height lower than *h* and prove its validity for trees of height *h*. Since *h* > 1, clause (2.4) must have been selected at the first step. Thus, **markingBefore(e_2,e_f)** and **beforeFact(e_1,e_2)** have successful derivations for some node **e_2**. By inductive hypothesis, the derivability of the former goal implies that there is a path *p*' = (**e_2**,**e_3**), ..., (**e_k**,**e_f**) between nodes **e_2** and **e_f**. Considering **(e_1,e_2)** together with *p*', we obtain the desired path.

We will now evaluate the cost of **before** as implemented by program 2, and show the impact on EC of substituting clauses (1.4-5) of program 1 with program 2. Let us first compute the cost of **markingBefore** and **unmarkAll**. It is easy to show that the latter is always called. Thus, the complexity of **before** is equal to the sum of the costs of the two.

The predicate **markingBefore** is designed to be called with both arguments instantiated. Since clause (2.4) marks a node before the recursive call and since the number of nodes (initially all unmarked) is *n*, this predicate is called at most *n* times. Moreover, each execution of **markingBefore** involves at most *n*-1 accesses to a **beforeFact** fact. Therefore, the overall cost for this predicate is $O(n^2)$. Since at most *n* nodes have been marked by **markingBefore**, **unmarkAll** has cost $O(n)$. Therefore, **before** costs at most $O(n^2)$. This upper bound is reached in the situation of figure 1.

After integrating this version of **before** into EC, the cost of **holds** becomes polynomial, dropping from $O(2^n)$ to $O(n^5)$. Indeed, if there are *k* events initiating *p* and *h* events terminating *p* in the database, the call to **happens(Ei)** in the body of **holds**, with **Ei** unbound, can succeed *n* times but **initiates(Ei,P)** will succeed only for *k* of the identified events. Analogously, **happens(Et)** succeeds *n* times but **terminates(Et,P)** retains only *h* events. As a result, $k \cdot h$ pairs of events are allowed to reach **before(Ei,Et)**. Since $k+h \leq n$, the product $k \cdot h$ is maximum for $k=h=n/2$, resulting in a quadratic number of pairs. For each pair, both **before(Ei,Et)** and **not broken(Ei,P,Et)** will be considered in the worst case. The cost of the former has been proved to be $O(n^2)$ above, while the cost of the latter is evaluated as follows: the first goal in **`broken`** is called with an uninstantiated argument and can succeed n times; for each success, at worst two calls to **`before`** ($2 \cdot O(n^2)$) and three constant cost calls (**`initiates`** and **`terminates`** with the first argument bound, and **`exclusive`**) are performed. Therefore, the cost of **`broken`** turns out to be cubic ($O(n) \cdot O(n^2)$). Returning to **`holds`**, we obtain a cost equal to $O(n^2) \cdot O(n^3) = O(n^5)$.

4 What the Event Calculus actually does

In this section, we aim at getting a better understanding of what EC actually does when computing MVIs in order to design more efficient versions. The representation of the ordering of events is of fundamental importance. Let $E=\{e_1, ..., e_n\}$ be the set of events, represented in EC by the predicate **`happens`**. The ordered pairs **`beforeFact(`**e_i**`,`** e_j**`)`** contained into the database constitute a set $w \subseteq E \times E$ that evolves gradually and incrementally from the empty set towards a total ordering relation. Notice however that the main axioms of EC never access w (i.e. **`beforeFact`** facts) directly. Instead, they rely heavily on the predicate **`before`** that models the transitive closure of w. Let us denote as w^+ the transitive closure of w. w^+ is a strict order on E, i.e. a relation that is irreflexive, asymmetric and transitive. w is a subset of w^+ and can indeed be viewed as a specification of it. It is easy to prove that w^+ can contain a quadratic number of edges; indeed the maximum number of edges $n \cdot (n-1)/2$ is reached when w^+ is a total order. From a graph-theoretic point of view, w corresponds to a directed acyclic graph G on E while w^+ corresponds to its completion G^+.

In order to compute the set of MVIs for a property p, the predicate **`holds`** in program 1 considers every pair of events e_i, e_j such that e_i initiates p and e_j terminates p, checks whether (e_i,e_j) belongs to w^+, i.e. if G^+ contains an edge from e_i to e_j, and ascertains that no interrupting event e occurs in between, i.e. that G^+ does not contain any node e associated to a property q such that p and q are exclusive and $(e_i,e) \in G^+$ and $(e,e_j) \in G^+$.

This approach presents two drawbacks. First, since G^+ is implicit in G, we showed in the previous section that the cost of checking whether an edge belongs to G^+ by means of the marking version of **`before`** is proportional to the number of edges present in G, i.e. to the number of **`beforeFact`** in the database. Second, EC blindly picks up every pair consisting of an event initiating p and an event terminating p, and only later looks for possible interruptions. We would like instead to include the determination of possibly interrupting events within the search of candidate MVIs for p (i.e. pairs (e_i,e_j) such that e_i initiates p and e_j terminates it). Both problems can be solved by shifting the emphasis from the transitive closure w^+ of w to its anti-transitive closure, or *kernel*, w^-. w^- is the least subset of w such that $(w^-)^+ = w^+$ and it can be obtained by removing every pair (e_i,e_j) from w such that $(e_i,e) \in w$ and $(e,e_j) \in w$ for some event e. The notion of kernel induces a subgraph G^- of G that does not contain any transitive edge. The number of edges in G^- is strictly lower than $n \cdot (n-1)/2$ and is indeed linear in most cases (as in the example reported in section 6). While experimenting, we have not been able to generate cases with more than $O(n\sqrt{n})$ edges.

5 How to do it efficiently

The results of section 3 call for optimization in those cases (we expect them to be the most frequent) where the critical operation is querying for the validity of a certain property. In these circumstances, the upper bound that comes out from the previous analysis is still not acceptable. Therefore, we will seek for optimized versions of EC having lower query cost.

We are interested in a solution that operates on the representation of ordering information in the database. We will first introduce the proposed technique in the case where a single property is involved. Then, the solution will be generalized in order to account for multiple (possibly exclusive) properties.

5.1 Storing and updating the kernel

In order to store and update the kernel of the ordering relation, clause (1.6) of program 1 must be replaced by the following program:

```
updateOrder(E1, E2) :-            (3.1)
   not before(E1, E2),
   assertOP(E1,E2).

assertOP(E1, E2) :-               (3.2)
   beforeFact(EA, EB),
   retractInBetween(E1,EA,EB,E2).
```

```
assertOP(E1, E2) :-                   (3.3)
   assert(beforeFact(E1, E2)), !.

retractInBetween(E1,EA,EB,E2) :-
   (E1=EA; before(EA, E1)), !,  (3.4)
   (EB=E2; before(E2, EB)), !,
   retract(beforeFact(EA, EB)),
   fail.
```

Program 3

Figure 2 shows the operations performed by each of these clauses. The dotted line represents the ordered pair that is considered for addition; the thin lines stand for single instances of **`beforeFact`**; finally, the thick lines represent sequences of zero or more chained instances of **`beforeFact`**.

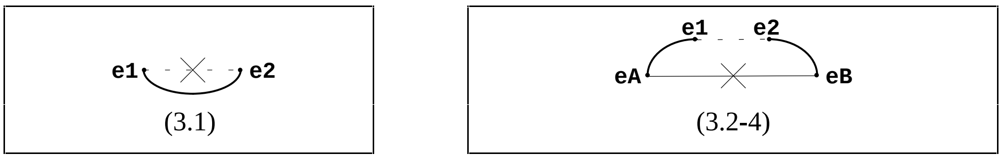


Figure 2

The leftmost schema illustrates the case where the added edge is already entailed by the current ordering relation. In order to maintain the minimality of the representation, the update must be ignored. This case is caught by clause (3.1) through the negative call to **`before`**.

The cooperating clauses (3.2-4) deal with the complementary case, i.e. when the added edge is not subsumed by the current ordering. It is then added by clause (3.3). Before doing this, edges becoming redundant because of transitivity must be individuated and retracted from the database. This can happen in the situation represented in the rightmost schema, and implemented by clauses (3.2) and (3.4). Adding the edge **`(e1,e2)`** can close a transitive relation between nodes **`eA`** (possibly **`e1`**) and **`eB`** (possibly **`e2`**). This is problematic when there exist already a direct link between these two nodes: this previously inserted ordered pair has now become redundant and must be removed. This is realized by clauses (3.2) and (3.4). Notice that there may exist several pairs of the above kind, and all of the corresponding edges must be retracted. This is achieved through backtracking by cohercing the failure of clause (3.4). When every possibility has been examined, the execution finally backtracks to clause (3.3) that succeeds asserting (only once) the added edge.

The cost of these operations is the following:

- **`retractInBetween(ea,e1,e2,eb)`**: when called ground, the cost of this predicate corresponds to the cost of two calls to **`before`** - $2O(n^2)$ - plus the constant cost of performing the retraction. Notice that no backtracking is allowed within this clause. The resulting cost is therefore quadratic.
- **`assertOP(e1,e2)`**: this predicate calls **`retractInBetween`** for each element of the kernel of the ordering relation (clause 3.2) and then asserts the ordered pair of interest (clause 3.3). We showed that an upper bound for the number of edges of the kernel is at most $O(n^2)$. Therefore, a call to this predicate can cost at most $O(n^4)$.
- **`updateOrder(e1,e2)`**: similarly to **`assertOP`**, this clause never fails. Its cost is given by the cost of one call to the negative goal, and one to **`assertOP`**, resulting in $O(n^4)$.

5.2 Single property

In the case of a single property *p*, once only the kernel of the ordering relation is retained in the database, clause (1.1) can be simplified as follows:

```
holds(period(Ei, p, Et)) :-
   happens(Ei), initiates(Ei, p),
   happens(Et), terminates(Et, p),
   beforeFact(Ei, Et).
```

Indeed, whenever *p* is the only property represented in the database, an event *e* interrupting a candidate MVI (e_i,e_j), where e_i initiates *p* and e_j terminates it, must either initiate or terminate *p*. Therefore (e_i,e_j) is an MVI for *p* if and only if e_i is an immediate predecessor of e_j in the current ordering, in the sense that no other event is recorded between the two. It is worth noting that the predicate **`broken`** is not needed in this case.

Computing the complexity of this restricted version of EC is trivial. In fact, the cost of **`holds(period(Ei,p,Et))`** consists in the two calls to **`happens`**. Since there are *n* events in the database, a call to **`holds`** costs $O(n^2)$. Finally, notice that a further improvement in efficiency (at least in the average case) can be obtained by eliminating the calls to **`happens`** and by rearranging the atomic goals in its body as follows:

```
holds(period(Ei, p, Et)) :-
   beforeFact(Ei, Et),
   initiates(Ei, p),
   terminates(Et, p).
```

The complexity of this clause is now equal to the number of **`beforeFact`** in the database, i.e. the cardinality of the kernel. As outlined in the section 4, this number is always less than $n\cdot(n\text{-}1)/2$ and linear in most cases.

5.3 Multiple properties

We now generalize the technique to the case of multiple properties. We maintain the minimality of the ordering information, as in the single property case, and implement a graph search algorithm for the query predicate **`holds`**. The resulting Prolog code, reported as program 4, will now be explained. **`holds`** (clause 4.1) starts the search from a specific initiating event. Let e_i be this event and p the corresponding property. If e_i is not instantiated in the query, all events in the graph will be processed. The idea is to examine all the successors of e_i searching for events terminating p. The search starts from its immediate successors, and proceeds breadth-first. Exhausting a layer before examining nodes in the next is indeed crucial for the soundness of the algorithm.

Clause (4.2) finds the elements in the first layer after e_i, and determines, by means of the predicate **`findTerminants`**, the correct terminating events for p among these nodes and their successors. **`findTerminants`** (clause 4.3) processes a layer. As depicted in Figure 3, it partitions the nodes of a layer in three categories: terminating events for p, events interfering with p (i.e. other initiating events for p, or for properties incompatible with p) and independent events. The predicate **`termP`** (clauses 4.4-6) is used to identify the events in the first category. Their successors are marked since there is no need to keep them into consideration during further processing. The nodes found by **`termP`** contribute to the solution returned to the user. Nodes in the second category are processed by means of the predicates **`initP`** together with the auxiliary predicate **`initPorEx`** (clauses 4.7-11). These events are simply marked as well as their successors since they cannot be contained in a successful path for the user query. The nodes that have passed through these two sieves are processed by the predicate **`otherP`** (clauses 4.12-13); they are used to determine the next layer to explore, which is obtained by collecting all of their unmarked immediate successors. The procedure repeats recursively until the most distant layer from e_i is examined.

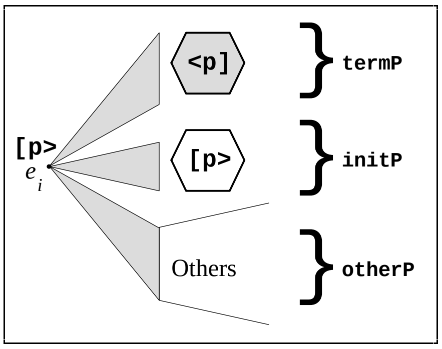


Figure 3

```
holds(period(Ei, P, Et)) :-          (4.1)
   happens(Ei, unmarked),
   initiates(Ei, P),
   findTerm(Ei, P, Et).
findTerm(Ei, P, Et) :-               (4.2)
   findSucc(Ei, Es),
   findTerminants(Es, P, Res),
   unmarkAll,
   !, member(Et, Res).
findTerminants(Es, P, Res) :-        (4.3)
   termP(P, Es, LessEs, ResTerm),
   initP(P, LessEs, FewerEs),
   otherP(P, FewerEs, ResOther),
   append(ResTerm, ResOther,Res).
```

```
termP(P, [E|Tail], NonTerm,
        [E|Term]) :-                 (4.4)
   terminates(E, P),
   markAll(E),
   termP(P, Tail, NonTerm, Term).
termP(P, [E|Tail], [E|NonTerm],
        Term) :-                     (4.5)
   not terminates(E, P),
   termP(P, Tail, NonTerm, Term).
termP(P, [], [], []).                (4.6)
initP(P, [E|Tail], NonP) :-          (4.7)
   initPorEx(E, P),
   markAll(E),
   initP(P, Tail, NonP).
initP(P, [E|Tail], [E|NonP]) :-  (4.8)
   not initPorEx(E, P),
   initP(P, Tail, NonP).
initP(P, [], []).                (4.9)
initPorEx(E, P) :-              (4.10)
   initiates(E, P).
initPorEx(E, P) :-              (4.11)
   (initiates(E,Q);terminates(E,Q)),
   exclusive(P, Q).
otherP(P, [E|Es], Res) :-       (4.12)
   listSucc([E|Es], SuccEs),
   findTerminants(SuccEs, P, Res).
otherP(P, [], []).              (4.13)
findSucc(E, Es) :-              (4.14)
   setof(NextE,
         (beforeFact(E, NextE),
         happens(NextE, unmarked)),
        Es), !.
findSucc(E, []).                (4.15)
listSucc([E|Es], SuccEEs) :-    (4.16)
   findSucc(E, SuccE),
   listSucc(Es, SuccEs),
   listUnion(SuccE, SuccEs,
   SuccEEs).
listSucc([], []).               (4.17)
markAll(E) :-                   (4.18)
   mark(E),
   findSucc(E, SuccE),
   markAllIn(SuccE).
markAllIn([E|Es]) :-            (4.19)
   markAll(E),
   markAllIn(Es).
markAllIn([]).                  (4.20)
listUnion([E|L1], L2, L3) :-    (4.21)
   member(E, L2), !,
   listUnion(L1, L2, L3).
listUnion([E|L1], L2, [E|L3]) :-
   listUnion(L1, L2, L3).       (4.22)
listUnion([], L, L).            (4.23)
```

Program 4

6 A sample execution

We will now analyze an example of EC with relative times and comment on the results of executing the basic and enhanced versions of EC. Consider a situation involving six events e_1, e_2, e_3, e_4, e_5 and e_6 referring to properties p and r, which are exclusive. The events e_1, e_5 and e_2, e_6 respectively initiate and terminate intervals of validity for r, while e_3 and e_4 are respectively the initiating and terminating event of p. The resulting Prolog representation for this factual knowledge is as follows:

```
happens(e1, unmarked).   initiates(e1, r).    exclusive(p, r).
happens(e2, unmarked).   terminates(e2, r).   exclusive(r, p).
happens(e3, unmarked).   initiates(e3, p).
happens(e4, unmarked).   terminates(e4, p).
happens(e5, unmarked).   initiates(e5, r).
happens(e6, unmarked).   terminates(e6, r).
```

These facts are intended for the marking implementation of EC, as it can be seen from the arity of the predicate **happens**. The Prolog code for the standard case is analogous and differs only for the absence of the second argument.

In the intended final ordering, events are ordered according to their indices. Therefore, the final situation is represented in figure 4.

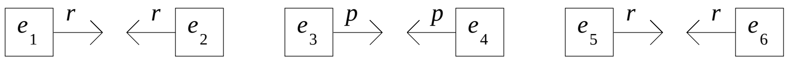


Figure 4

The ordering information is input in arbitrary order. In our example, we will consider the following sequence of ordered pairs, which arrive one at a time:

(e_1, e_4) - (e_1, e_6) - (e_2, e_4) - (e_1, e_2) - (e_3, e_4) - (e_4, e_5) - (e_2, e_3) - (e_2, e_6) - (e_5, e_6)

Notice that the transitive closure of the final ordering is indeed the order relation represented in figure 4. These ordered pairs are entered into the Prolog database by running the following goals, in sequence.

```
?- updateOrder(e1,e4).  | ?- updateOrder(e1,e2).  | ?- updateOrder(e2,e3).
?- updateOrder(e1,e6).  | ?- updateOrder(e3,e4).  | ?- updateOrder(e2,e6).
?- updateOrder(e2,e4).  | ?- updateOrder(e4,e5).  | ?- updateOrder(e5,e6).
```

Table 1 shows the evolution of the computation: each row corresponds to the addition of one of these ordered pairs to the database. The first column shows which update is being performed. The second column gives a visual account of the content of the database. In particular the kernel is represented in solid lines while derivable edges are drawn as dotted lines. The third column contains the list of the MVIs deduced by EC, i.e. the result of running a generic query of the form **`?- holds(X)`**. For conciseness, we represented **`period(ei,q,et)`** for a property **`q`** as the more compact **`q(ei,et)`**. Columns 6 to 8 and 7 to 11 contain the results of our experiments for the basic (program 1) and enhanced (programs 2, 3 and 4) version of EC respectively. In both cases, the cost is measured as the number of nodes visited in the resolution tree. The column headed as *Q.1* indicates the number of nodes that have been explored to obtain the first solution to the generic query **`?- holds(X)`**. The following column, *Q. Fail*, show the cost of an exhaustive query, resulting from keeping on asking for more solutions until the computation stops. Finally, the column *Update* shows the cost of asserting the ordered pair in the first column. Remember that the enhanced algorithm for EC maintains the kernel of the ordering relation and is implemented as shown in program 3, rather than by simply adding the fact corresponding to the new ordered pair.

The experimental data reported in Table 1 show that the enhanced EC is more efficient than the basic EC in the query phase. This fact becomes more and more evident as the number of ordered pairs into the knowledge base grows: if the enhanced EC is just 17.5% faster (40 nodes analyzed versus 47) when there is no ordering information, after adding the last ordered pair, the first answer is retrieved 322% faster (86 nodes examined instead of 363) and the basic implementation explores 399% nodes more (889 versus 178) in the case of the exhaustive query. Of course, the update operation is more expensive in the enhanced case. Just one node is explored in the basic implementation. This extra-cost is acceptable considering the benefits produced in the query phase and the fact that the overall cost of the enhanced version is anyway lower.

8 Conclusions

The paper analyzed in detail the process of computing maximal validity intervals for properties in Kowalski and Sergot's Event Calculus, and proposed a revision of the calculus that strongly increases its efficiency when dealing with partial ordering information. The resulting calculus models events and their ordering relations in terms of a directed acyclic graph, and incorporates a marking technique to speed up the visit of the graph during the computation of validity intervals. Moreover, it provides an alternative solution to the problem of supporting default persistence that further improves its performance. Instead of the expensive generate-and-test approach of the original calculus, it restricts a priori the search space by exploiting the kernel of an ordering relation.

Besides the procedural Prolog encoding presented in this paper, these ideas have been given a declarative implementation by means of the linear logic programming language Lolli [6]. The choice of this language was based on its ability to provide a declarative substitute to Prolog's **`assert`** and **`retract`** by means of implication goals and resource consumption mechanism typical of linear logic, respectively. Moreover, all the other extra-logical features of Prolog used in this paper can be easily coded in Lolli.

References

[1] I. Cervesato, A. Montanari, A. Provetti: "On the Non-monotonic Behavior of Event Calculus for Deriving Maximal time Intervals", *International Journal of Interval Computation*, 1993, pp. 83-119.

[2] I. Cervesato, L. Chittaro, A. Montanari: "Modal Event Calculus in Lolli", submitted, 1994.

[3] L. Chittaro, A. Montanari: "Reasoning about Discrete Processes in a Logic Programming Framework", in *Proc. GULP'93 - Eight Conference on Logic Programming*, Gizzeria Lido (CZ), Italy, June 1993, pp. 407-421.

[4] L. Chittaro, A. Montanari: "Efficient Handling of Context Dependency in the Cached Event Calculus", in *Proc. TIME'94 - International Workshop on Temporal Representation and Reasoning*, Pensacola, FL, USA, 1994, pp. 103-112.

[5] L. Chittaro, A. Montanari, A. Provetti: "Skeptical and Credulous Event Calculi for Supporting Modal Queries", in *Proc. ECAI'94 - 11th European Conference on Artificial Intelligence*, Amsterdam, The Netherlands, August 1994, pp. 361-365.

[6] J.S. Hodas, D. Miller: "Logic Programming in a Fragment of Linear Logic", to appear in the *Journal of Information and Computation*.

[7] R. Kowalski: "Database Updates in the Event Calculus", in *Journal of Logic Programming*, Vol. 12, June 1992, pp 121-146.

[8] R. Kowalski, M. Sergot: "A Logic-based Calculus of Events", in *New Generation Computing*, Vol. 4, Ohmsha Ltd and Springer-Verlag, 1986, pp 67-95.

[9] A. Montanari, E. Maim, E. Ciapessoni, E. Ratto: "Dealing with Time Granularity in the Event Calculus", in *Proc. of FGCS'92*, Tokyo, Japan, June 1992, pp 702-712.

[10] M. Sergot: "*(Some Topics in) Logic Programming in AI*", Lecture Notes of the GULP Advanced School on Logic Programming, Alghero, Italy, 1990.

[11] S. Sripada: "A Metalogic Programming Approach to Reasoning about Time in Knowledge Bases", in *Proc. of IJCAI'93 - 13th International Joint Conference on Artificial Intelligence*, Chambery (France), pp 860-865.

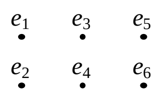
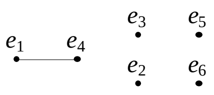
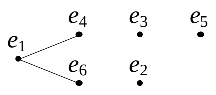
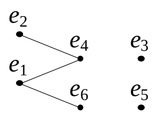
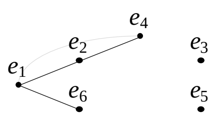
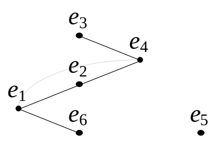
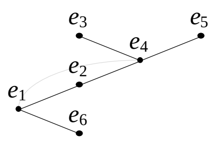
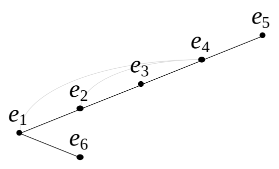
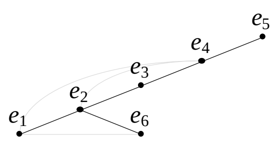
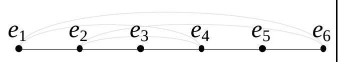
w	w visually	X: holds(X)	Basic EC			Enhanced EC		
			Q. 1	Q. Fail	Update	Q. 1	Q. Fail	Update
			47	47	–	40	40	–
+ (e_1, e_4)			53	53	1	60	60	10
+ (e_1, e_6)		$r(e_1, e_6)$	85	122	1	36	64	31
+ (e_2, e_4)		$r(e_1, e_6)$	85	122	1	51	69	78
+ (e_1, e_2)		$r(e_1, e_2)$ $r(e_1, e_6)$	117	287	1	53	57	86
+ (e_3, e_4)		$r(e_1, e_2)$ $r(e_1, e_6)$ $p(e_3, e_4)$	117	322	1	53	101	125
+ (e_4, e_5)		$r(e_1, e_2)$ $r(e_1, e_6)$ $p(e_3, e_4)$	168	473	1	75	137	110
+ (e_2, e_3)		$r(e_1, e_2)$ $r(e_1, e_6)$ $p(e_3, e_4)$	249	653	1	89	149	177
+ (e_2, e_6)		$r(e_1, e_2)$ $p(e_3, e_4)$	273	529	1	87	146	133
+ (e_5, e_6)		$r(e_1, e_2)$ $p(e_3, e_4)$ $r(e_5, e_6)$	363	889	1	86	178	166

Table 1