Speeding up temporal reasoning by exploiting the notion of kernel of an ordering relation

Luca Chittaro, **Angelo Montanari**
Dipartimento di Matematica e Informatica
Università di Udine, Via delle Scienze, 206
33100 Udine - ITALY
{chittaro\montana}@dimi.uniud.it

Iliano Cervesato
Dipartimento di Informatica
Università di Torino, Corso Svizzera, 185
10149 Torino - ITALY
iliano@di.unito.it

1 Introduction

In this paper, we consider the problem of expediting temporal reasoning about partially ordered events in Kowalski and Sergot's Event Calculus (EC). EC is a formalism for representing and reasoning about events and their effects in a logic programming framework [7]. 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). Database updates in EC provide information about the occurrences of events and their times [6] and are of additive nature only. We assume here that the set of events is fixed, and the input process consists in the addition of ordering information.

We will show how the introduction of partial ordering heavily increases the computational complexity of deriving MVIs. Then, we will provide a precise characterization of what EC actually does to compute MVIs, and propose a solution to do it efficiently when only incomplete information about event ordering is available.

The paper is organized as follows. In Section 2, we introduce the basic features of EC with relative times and partial ordering. In Section 3, we analyze its computational complexity, that turns out to be exponential. Moreover, we show how complexity can be reduced to polynomial ($O(n^5)$) by adopting a graph marking technique that speeds up search. In Section 4, we provide a deeper analysis of how EC derives MVIs, and we formally introduce the notion of kernel of an ordering relation. Then, we show how the notion of kernel can be usefully applied to further reduce the complexity of computing MVIs. Section 5 discusses an example, also contrasting the set of MVIs with the sets of necessarily and possibly true MVIs. Section 6 concludes the paper.

2 The Event Calculus with relative times and partial ordering

EC 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.

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* of type *tye* by means of the clause:

```
happens(e,tye).
```

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

```
initiates(tye, p1).       terminates(tye, p2).
```

The **initiates** (**terminates**) clause relates each type of event *tye* to the property *p* it initiates (terminates).

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

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

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

The **holds** 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 **broken** axiom states that a given property P ceases to hold if there is an event E that happens between Ei and Et and initiates or terminates a property Q that is exclusive with P. The **exclusive**(P, Q) predicate is a constraint to force the derivation of P to fail when it is possible to conclude that Q holds at the same time. Finally, the condition $P=Q$ constrains interferences due to incomplete sequences of events relating to the same property. It indeed guarantees 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 ending P respectively. The **exclusive** facts have obviously to be defined for each specific application (e.g. **exclusive(p,q).**). 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):-                          (1.3)
    beforeFact(E1,E2).

before(E1,E2):-                          (1.4)
    beforeFact(E1,E3),before(E3,E2).
```

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

```
updateOrder(E1, E2) :-                   (1.5)
    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 (1.1-5), 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 [3]. In this section, a worst case complexity analysis will be carried out for EC with relative times and partial ordering. 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 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(Ei,TyEi)` and `happens(Et,TyEt)`: each of this goal succeeds n times, since n events are recorded in the database. So, the cost of each is $O(n)$.

`initiates(TyEi,P)` and `terminates(TyEt,P)`: the cost of these predicates is constant for a ground `TyEi` (or `TyEt`) even when they are called with `P` uninstantiated (as in clause 1.2).

`exclusive(P,Q)`: this predicate is always called ground and it thus can be matched against at most one fact in the database. The cost is therefore constant.

`beforeFact(E1,E2)`: When called ground, as in clause (1.3), the query cost is constant. In clause (1.4) 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(Ei,Et)`: we will show that the standard two-clauses definition of `before` (clauses 1.3-4) 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 (`en-1` and `en`) participate in a total order between `e1` and `en-2`. All the transitive pairs, but the pair `(e1,en)` are explicitly specified by means of `beforeFact`. We assume that `beforeFact(e1,en-1)` textually follows any other fact of the form `beforeFact(e1,ei)`, for `i = 2..n-2`.

Due to the operational behavior of PROLOG, EC thus tries to prove `before(e1,en)` looking for a path that passes through `en-2` before attempting the (only possible) path that includes `en-1`.
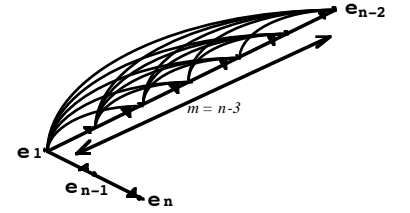

Figure 1

Backtracking due to the failure in proving `before(en-2,en)` causes every path from `e1` to `en-2` to be unsuccessfully attempted. The resulting cost is computed as follows. Let $m=n$-3 be the length of the longest path between `e1` and `en-2`. We have the following recursive relation, where $C_{bf}(m)$ 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"` (the value of m highlighted in Figure 1 concerns the pair `(e1,en-2)`):

$$C_{bf}(1) = 1; \qquad\qquad m=1$$

$$C_{bf}(m) = m + \sum_{i=1}^{m-1} C_{bf}(i) \qquad m>1$$

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 1.3). If $m>1$, each of the remaining m-1 edges leads to a node `e` (clause 1.4), 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 $C_{bf}(m)$, let us unfold the expression for $C_{bf}(m)$:

$$C_{bf}(m)=m+C_{bf}(m\text{-}1)+ C_{bf}(m\text{-}2) + C_{bf}(m\text{-}3) +... +C_{bf}(1)=$$
$$=m + (m\text{-}1)+2C_{bf}(m\text{-}2) +2C_{bf}(m\text{-}3) +... +2C_{bf}(1)=$$
$$=m+ (m\text{-}1) + 2(m\text{-}2) +4C_{bf}(m\text{-}3) +... +4C_{bf}(1)=$$
$$=m+ 2^0(m\text{-}1)+ 2^1(m\text{-}2) + 2^2(m\text{-}3) +... +2^{m\text{-}2}(1)$$

We can summarize this formula as:

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

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 times 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 foresee 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.

```
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, TyE, marked)),
   assert(happens(E, TyE, unmarked)).
mark(E) :-                                  (2.8)
   retract(happens(E, TyE, unmarked)),
   assert(happens(E, TyE, marked)).
happens(E,TyE):-                            (2.9)

   happens(E,TyE,_).
```

Program 2

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 third argument contains either **marked** or **unmarked** with the obvious meaning. Initially all the nodes are unmarked. The purpose of clause (2.9) is 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 (1.3-4) yields **before(e',e")** if and only if the database contains a path leading from **e'** to **e"**, 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 **markingBefore(e',e")**, and consequently **before(e',e")**.

First suppose that the database contains at least a path $p=(e',e_1),(e_1,e_2),...,(e_{k-1},e_k), (e_k,e")$ of length $k+1$ that links nodes **e'** and **e"** passing through k intermediate nodes 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)). If there is more than one path from **e'** to **e"**, let p be that path whose first edge comes first in the listing of **beforeFact** (if there is more than one path from **e'** to **e"** with this edge as its first edge, then let p be the path whose second edge comes first in the listing of **beforeFact**; and so on). In other words, this selected path is the first path from **e'** to **e"** considered by the PROLOG control strategy. Let us prove that **markingBefore(e',e")** is derivable from program 2, and only nodes $e_1,e_2,..,e_k$ of the selected path are marked during this process. The proof is inductive in the length of the selected path. The case where the length of the path is 1, i.e. there are 0 intermediate nodes and $p=(e',e")$, is caught by clause (2.3). We assume, as inductive hypothesis, that the statement holds for length k (i.e. $k-1$ intermediate nodes), and we prove that it holds for length $k+1$ (i.e. k intermediate nodes). Let us consider an instance of clause (2.4) where $E1=e'$ and $E2=e"$. The first subgoal matches **beforeFact(e',e_1)** in the database, instantiating **E3** to e_1. By hypothesis, all nodes are initially unmarked and e_1 can not have been marked up to now: if e_1 were marked, then it would have been already reached by traversing another path, but this would contradict the hypothesis that p (to which e_1 belongs) is the first path from **e'** to **e"** considered by the PROLOG control strategy. Therefore, the second subgoal, **happens(e_1,_,unmarked)**, is immediately provable. Moreover, the presence of this fact in the database causes the subgoal **mark(e_1)** to mark e_1 by clause (2.8). By inductive hypothesis, **markingBefore(e_1,e")** is derivable and nodes $e_2,..,e_k$ of the selected path are marked as a by-product of the process. Thus, the overall clause succeeds and proves the goal **markingBefore(e',e")**.

Conversely, suppose that **markingBefore(e', e")** is derivable from program 2. We proceed by induction on the height h of the resolution tree for **markingBefore(e',e")**. If $h=1$, then clause (2.3) has been applied, and the database contains the fact

`beforeFact(e',e")`. 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, `beforeFact(e',e1)` and `markingBefore(e1,e")` have successful derivations for some node `e1`. By inductive hypothesis, the derivability of the former goal implies that there is a path $p'=$`(e1,e2)`,...,`(ek,e")` between nodes `e1` and `e"`. Considering `(e',e1)` 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 (1.3-4) 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,TyEi)` in the body of `holds`, with `Ei` unbound, can succeed n times but `initiates(TyEi,P)` will succeed only for k of the identified events. Analogously, `happens(Et,TyEt)` succeeds n times but `terminates(TyEt,P)` retains only h events. As a result, $k·h$ pairs of events are allowed to reach `before(Ei,Et)`. Since $k+h \le n$, the product $k·h$ 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` ($2·O(n^2)$) 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 ($O(n)·O(n^2)$). Returning to `holds`, we obtain a cost equal to $O(n^2)·O(n^3)=O(n^5)$.

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

In this section, we aim at getting a better understanding of what EC actually does when computing MVIs. This insight allows to design more efficient versions of EC. The representation of the ordering of events is of primary 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 $w \subseteq E \times E$. 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 $n·(n-1)/2$ 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, whose nodes are event occurrences and 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., 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 exclusive with p (or associated to the property p itself) such that $(e_i,e) \in G^+$ and $(e,e_j) \in G^+$.

This approach presents two drawbacks. First, 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). Second, 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 in G, i.e. to the number of `beforeFact` in the database. 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 $n·(n-1)/2$ and is indeed linear in most cases (as in the example reported in Section 5).

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. The solution we propose in the following 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 properties.

4.1 Storing and updating the kernel

In order to store and update the kernel of the ordering relation, clause (1.5) of program 1 must be replaced with 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) :-          (3.4)
    (E1=EA; before(EA, E1)), !,
    (EB=E2; before(E2, EB)), !,
    retract(beforeFact(EA, EB)), fail.
```

Program 3

When the edge **(e1,e2)** is already entailed by the current ordering relation, it is not added to the representation, in order to maintain minimality. This case is caught by clause (3.1) through the negative call to **before**.

Co-operating clauses (3.2-4) deal with the complementary case, i.e. edge **(e1,e2)** is not subsumed by the current ordering. Edge **(e1,e2)** is added to the representation by clause (3.3). Before doing this, edges becoming redundant because of transitivity must be located and retracted from the database. As shown by figure 2 (where the thin lines represent sequences of zero or more chained instances of **beforeFact**), adding the edge **(e1,e2)** can close a transitive relation between nodes **eA** (possibly **e1**) and **eB** (possibly **e2**). This is problematic when there exists already a direct link between these two nodes: this previously inserted link has now become redundant and must be removed. This is done by clauses (3.2) and
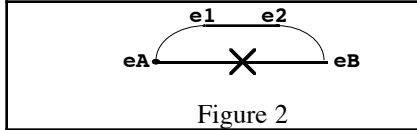

Figure 2

(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 forcing 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(E1,EA,EB,E2): this predicate is called ground and its cost thus corresponds to the cost of two ground 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): this predicate is called ground and its cost is given by the cost of one negative call to **before**, and one to **assertOP**, resulting in $O(n^4)$ complexity.

4.2 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,TyEi), initiates(TyEi, p),
    happens(Et,TyEt), terminates(TyEt, 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),
    happens(Ei,TyEi), initiates(TyEi, p),
    happens(Et,TyEt), terminates(TyEt, p).
```

The complexity becomes equal to the number of **beforeFact** in the database, i.e. the cardinality of the kernel.

4.3 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**. This algorithm is implemented by the program given in the Appendix, where **holds** is clause (4.1).

The search is started 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.

As depicted in Figure 3, the nodes in the first layer after e_i, are partitioned 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 nodes in the first category are terminating events in the MVIs returned to the user, and their successors are marked since there is no need to keep them into consideration during further processing. Nodes in the second category are simply marked as well as their successors since they cannot be contained in a successful path for the user query. The nodes in the third category are used to determine the next layer to explore,

Figure 3

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. With reference to the code in the Appendix, **findTerm** (clause 4.2) finds the elements in the first layer after a given node, **findTerminants** (clause 4.3) is the predicate which processes a layer and partitions the nodes, **termP** (clauses 4.4-6) is used to identify the events in the first category, nodes in the second category are processed by means of the predicates **initP** together with the auxiliary predicate **initPorEx** (clauses 4.7-11), and the remaining nodes are processed by the predicate **otherP** (clauses 4.12-13).
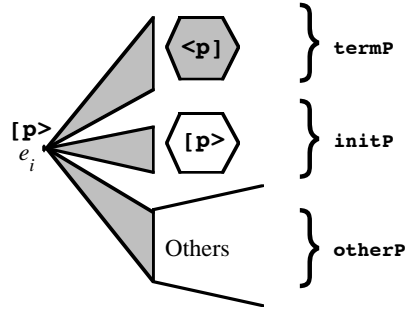
5 Analyzing the beverage dispenser example

We will now introduce an example, comment on the results of executing the basic and enhanced versions of EC on it, and further characterize the set of MVIs computed on this example by contrasting it with the set of possibly true and the set of necessarily true MVIs, respectively.

Consider the functioning of the simple beverage dispenser depicted in Figure 4: by setting the selector to the apple or to the orange position, apple juice or orange juice is obtained, respectively. Choosing the stop position terminates the output of juice. We model this knowledge as follows:

```
initiates(selectApple,supplyApple).
initiates(selectOrange,supplyOrange).
terminates(selectStop,  supplyApple).
terminates(selectStop,  supplyOrange).
exclusive(supplyApple,supplyOrange).
exclusive(supplyOrange,supplyApple).
```

We consider an actual situation where six events happened: e_1, e_2, e_3, e_4, e_5, and e_6. The following PROLOG factual knowledge associates events to their types:

```
happens(e1,selectApple,unmarked).
happens(e2,selectStop,unmarked).
happens(e3,selectOrange,unmarked).
happens(e4,selectStop,unmarked).
happens(e5,selectApple,unmarked).
happens(e6,selectStop,unmarked).
```

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 third argument. In the intended final ordering, events are ordered according to their indices. Therefore, the final situation is represented in figure 5. 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).
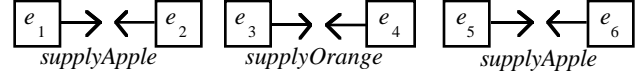

Figure 5

This sequence has been devised so that the complete situation shown in Figure 5 can be fully derived only after the last update. The 9 ordered pairs are entered into the PROLOG database by running the following goals, in sequence.

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

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 deleted edges are drawn as dotted lines. The third column contains the list of the MVIs derived by EC, i.e. the result of running a generic query of the form **?- holds(X).** For conciseness, we represented **period(ei,supplyApple,et)** as the more compact **a(ei,et)** and **period(ei,supplyOrange,et)** as the more compact **o(ei,et)**.
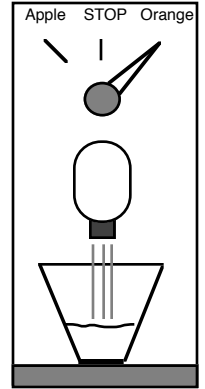

Figure 4

The experimental data (number of nodes visited in the resolution tree) obtained with this example show a more efficient behavior for the enhanced version of 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 only slightly faster (40 nodes analyzed versus 47) when there is no ordering information, after adding the last ordered pair the first answer is retrieved 4.2 times faster (86 nodes examined instead of 363) and the basic implementation explores 5 times more nodes (889 versus 178) to find all the MVIs. Of course, the update operation is more expensive in the enhanced case, since just one node is explored in the basic implementation. Anyway, this extra-cost is acceptable considering the benefits produced in the query phase and the fact that the overall cost (update and query) of the enhanced version is lower.

w	w visually	X: holds(X)	w	w visually	X: holds(X)
	(diagram: e_1 e_3 e_5 / e_2 e_4 e_6)		$+(e_3,e_4)$	(diagram)	$a(e_1,e_2)$ $a(e_1,e_6)$ $o(e_3,e_4)$
$+(e_1,e_4)$	(diagram)		$+(e_4,e_5)$	(diagram)	$a(e_1,e_2)$ $a(e_1,e_6)$ $o(e_3,e_4)$
$+(e_1,e_6)$	(diagram)	$a(e_1,e_6)$	$+(e_2,e_3)$	(diagram)	$a(e_1,e_2)$ $a(e_1,e_6)$ $o(e_3,e_4)$
$+(e_2,e_4)$	(diagram)	$a(e_1,e_6)$	$+(e_2,e_6)$	(diagram)	$a(e_1,e_2)$ $o(e_3,e_4)$
$+(e_1,e_2)$	(diagram)	$a(e_1,e_2)$ $a(e_1,e_6)$	$+(e_5,e_6)$	(diagram)	$a(e_1,e_2)$ $o(e_3,e_4)$ $a(e_5,e_6)$

Table 1

	MVIs derived by EC	Necessary MVIs	Possible MVIs
	$\varnothing$	$\varnothing$	a(e1,e2),a(e1,e6),o(e3,e4),a(e5,e2),a(e5,e6)
?- updateOrder(e1,e4).	$\varnothing$	$\varnothing$	a(e1,e2),a(e1,e6),o(e3,e4),a(e5,e2),a(e5,e6)
?- updateOrder(e1,e6).	a(e1,e6)	$\varnothing$	a(e1,e2),a(e1,e6),o(e3,e4),a(e5,e2),a(e5,e6)
. . .	. . .	. . .	. . .
?- updateOrder(e2,e3).	a(e1,e2),a(e1,e6),o(e3,e4)	$\varnothing$	a(e1,e2),a(e1,e6),o(e3,e4),a(e5,e6)
?- updateOrder(e2,e6).	a(e1,e2),o(e3,e4)	a(e1,e2)	a(e1,e2),o(e3,e4),a(e5,e6)
?- updateOrder(e5,e6).	a(e1,e2),o(e3,e4),a(e5,e6)	a(e1,e2),o(e3,e4),a(e5,e6)	a(e1,e2),o(e3,e4),a(e5,e6)

Table 2

Further insights on the behavior of EC with partially ordered events can be obtained by comparing it with the behavior of the Skeptical EC and Credulous EC, two variants of EC we proposed [4] to compute the MVIs which are derivable in all the total orders consistent with the given partial order (Skeptical EC), and those derivable in at least one of the total orders (Credulous EC). As an example, Table 2 considers again the beverage dispenser example and provides a comparison of the MVIs computed by the calculus presented in this paper, contrasted with those computed by the Skeptical and the Credulous calculus. Dean and Boddy [5] studied the task of deriving which facts must be or can possibly be true over certain intervals of time in presence of partially ordered events (in our context, which MVIs must

necessarily hold and which possibly hold), focusing on the computational complexity of the task and showing that it is intractable in the general case. In [2,4], we focused on providing the task with a modal logic formulation. While [5] develop polynomial algorithms to compute supersets of the set of possible facts and subsets of the set of necessary facts, the calculus presented in this paper polynomially computes a set in between, mimicking some behaviors of human reasoners.

6 Conclusions

The paper analyzed in detail the process of computing MVIs for properties in EC, 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.

Since we did not determine a lower bound for the complexity of the problem of deriving MVIs with partially information about event ordering, the possibility of further improving the achieved results can not be excluded.

References

[1] I. Cervesato, A. Montanari, A. Provetti 1993. "On the Non-monotonic Behavior of Event Calculus for Deriving Maximal Time Intervals", *Interval Computations* 2, 83-119.

[2] I. Cervesato, L. Chittaro, A. Montanari 1995. "A Modal Calculus of Partially Ordered Events in a Logic Programming Framework". *Proc. ICLP '95*, Tokyo, Japan, MIT Press.

[3] L. Chittaro, A. Montanari 1996. "Efficient temporal reasoning in the Cached Event Calculus", to appear in *Computational Intelligence Journal*.

[4] L. Chittaro, A. Montanari, A. Provetti 1994. "Skeptical and Credulous Event Calculi for Supporting Modal Queries", in *Proc. ECAI '94*, Amsterdam, The Netherlands, Wiley and Sons Publishers, 361-365.

[5] T. Dean, M. Boddy 1988. "Reasoning about Partially Ordered Events", *Artificial Intelligence* 36, 375-399.

[6] R. Kowalski 1992. "Database Updates in the Event Calculus", in *Journal of Logic Programming* 12, 121-146.

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

Appendix

```
holds(period(Ei, P, Et)) :-                    (4.1)
   happens(Ei, TyEi, unmarked),
   initiates(TyEi, 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)
   happens(E,TyE),terminates(TyE, P),
   markAll(E),termP(P, Tail, NonTerm, Term).
termP(P,[E|Tail],[E|NonTerm],Term) :-          (4.5)
   happens(E,TyE),
    not terminates(TyE, 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)
   happens(E,TyE),
   initiates(TyE, P).
initPorEx(E, P) :-                             (4.11)
   happens(E,TyE),
   (initiates(TyE,Q);terminates(TyE,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]) :-               (4.22)
   listUnion(L1, L2, L3).
listUnion([], L, L).                           (4.23)
```