Efficient Handling of Context-Dependency in the Cached Event Calculus

Luca Chittaro and **Angelo Montanari**
Dipartimento di Matematica e Informatica, Università di Udine,
Via Zanon, 6, 33100 Udine, Italy
e-mail: {chittaro | montanari}@uduniv.cineca.it

Abstract

This paper deals with the problem of providing temporal deductive databases with an efficient implementation in a logic programming framework. We restrict our attention to historical databases based on Kowalski and Sergot's Event Calculus extended with context-dependency. The paper aims at being beneficial to both the theoretically-minded and the implementation-oriented research communities. It provides a mathematical analysis of the computational complexity of query and update processing in the Event Calculus, and proposes a cached version of the calculus that moves computational complexity from query to update processing, and features an absolute improvement of performance when context-dependency is added.

1 Introduction

The paper deals with the problem of providing temporal deductive databases (TDDs) with an efficient implementation in a logic programming framework. TDDs provide the possibility of managing temporal information not only to retrieve information as it was stored in the database, but also to automatically derive further data [1, 2, 14, 20]. In the case of incomplete temporal data, information that is neither explicitly asserted nor monotonically implied by the available knowledge can be inferred by drawing conclusions according to suitable assumptions such as closed world or default persistence. Conclusions derived in this way are obviously defeasible and TDDs must withdraw them if the addition of further information makes them inconsistent. Different TDDs have been proposed in the literature [9, 11, 12, 16]. They differ from each other either in the underlying model of change (state-based in [15, 16], event-based in [11, 12]), or in the programming language paradigm they adopt (functional programming in [9, 10], logic programming in [11, 12]), or in both. We restrict our attention to historical databases based on Kowalski and Sergot's Event Calculus model of change enriched with context-dependency [5, 11].

From a description of events that occur in the real world, the Event Calculus (EC) allows one to derive various properties and the time periods for which they hold. The addition of context-dependency makes it possible to constrain the initiation and termination of properties to the validity of some given conditions at the time at which events occur. EC database and rules are formulated in a logic programming framework as a logic program. EC presents several advantages over relational temporal databases as well as over most TDDs [12]. First of all, while relational temporal databases only record the starting and ending of properties, thus losing the semantical structure of causing events, EC supports an explicit representation of events. In this way, updates are performed by entering events and deriving the starting and ending of properties as a logical consequence from event descriptions by means of general rules which express the semantics of events. Moreover, the domain modeler is fully in charge of ensuring the integrity of data, and thus expensive integrity checking procedures are not needed. Secondly, insertion of events in the database is not required to follow the chronological order of their occurrences. Thirdly, default persistence can be defined both in the future and the past.

The cost of update processing in EC is the cost of entering a new event, and thus it is constant [11], while query processing is expensive. The addition of context-dependency, which is crucial to deal with real-world problems [3, 5, 8], heavily deteriorates performance of query processing. The paper extends EC with a *lemma storage* mechanism (Cached Event Calculus) that records maximal validity intervals (MVIs) of properties for later use in query processing and updates them as soon as a new event is entered in the database. CEC moves computational complexity from query to update processing, and features an absolute improvement of performance when context-dependency is added, because CEC update processing costs less than EC query processing. Furthermore, CEC preserves the basic requirement of EC of making no assumptions about the temporal order of input events. As an example, CEC accepts an event happened before some of the already acquired events and revises all and only the affected MVIs. Obviously, acquiring a complete sequence of events according to their chronological order avoids the revision of past data, thus strongly increasing the performance of CEC. This is the case of several application domains, such as patient monitoring [8].

The paper is organized as follows. Section 2 introduces EC extended with context-dependency. Section 3 presents in detail the basic features of CEC. It illustrates how entering an event in the database may cause the assertion of new MVIs for the properties it affects or the shortening of cached ones, and how assertions and retractions are possibly propagated to other properties. A sample execution of CEC concludes the section. Section 4 analyzes and compares the complexity of query and update processing in EC and CEC. Conclusions provide an assessment of the work.

2 The logical calculus of events

In this section, we present the main features of the logical calculus of events. It extends EC with the notions of type and context-dependency. EC proposes a general approach to represent and reason 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 also 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 an *history* and is modeled by a set of event occurrences (event instances). The calculus allows us to infer a set of time intervals over which the properties initiated and/or terminated by event occurrences maximally hold (property instances). *Instances* of events and properties are obtained by attaching a time-point and a time-interval to event and property types and are denoted by the pairs *(event,time-point)* and *(property, time-interval)*, respectively[1]. Formally, we represent event occurrences by means of the *happens_at* predicate:

```
happens_at(event,timePoint).
```

Furthermore, we represent (part of the) domain knowledge by means of *initiates_at* and *terminates_at* predicates that express the effects of events on properties:

```
initiates_at(event1,prop1,T):-
  happens_at(event1,T).

terminates_at(event2,prop2,T):-
  happens_at(event2,T)
```

Initiates_at (*terminates_at*) predicates state that each instance of *event1* (*event2*) initiates (terminates) a period of time during which *prop1* (*prop2*) holds, respectively[2]. A particular *initiates_at* clause can be used to deal with *initial conditions*. Initial conditions describe a possibly partial initial state of the world and are specified by means of a number of events of type *initially(prop)*. Their validity from the beginning of time can then be derived by means of the clause:

```
initiates_at(initially(Prop),Prop,0):-
  happens_at(initially(Prop),0).
```

Such a clause is parametric with respect to the property argument and takes 0 as the initial instant of the time axis. This allows us to distinguish explicitly stated initial conditions (starting at 0) from initial conditions derived by persistence in the past, and to assign a greater degree of confidence to the former.

2.1 The basic axioms of EC

The basic Event Calculus model of time and change is defined by means of a set of axioms. The first axiom we introduce is the *mholds_for*. It allows us to state that the property P holds maximally (i.e. there is no larger time-interval for which it also holds) over $[Start, End]$ if an event *E1* which initiates P occurs at the time *Start*, and an event *E2* which terminates P occurs at time *End*, provided there is no known interruption in between:

```
mholds_for(P,[Start,End]):-
  initiates_at(E1,P,Start),
  terminates_at(E2,P,End),
  End gt Start, \+ broken_during(P,[Start,End]).

mholds_for(P,[Start,infPlus]):-
  initiates_at(E1,P,Start),
  \+ broken_during(P,[Start,infPlus]).

mholds_for(P,[infMin,End]):-
  terminates_at(E2,P,End),
  \+ broken_during(P,[infMin,End]).
```

where the predicate *gt* extends the ordinary ordering relationship $>$ to include the cases involving infinite arguments, sintactically denoted by *infMin* and *infPlus*. Analogously, we defined the predicates *ge, lt* and *le* which extend $\geq, <$ and $\leq$, respectively. The negation involving the *broken_during* predicate is interpreted using negation-as-failure. This means that properties are assumed to hold uninterrupted over an interval of time on the basis of failure to determine an interrupting event. Should we later record an initiating or terminating event within this interval, we can no longer conclude that the property holds over the interval. This gives us the non-monotonic character of the calculus which deals with default persistence. The predicate *broken_during* is defined as follows:

[1]The pair *(event, time-point)* uniquely identifies an event occurrence provided that two events of the same type can not simultaneously happen. In situations where this assumption is not acceptable, explicit identifiers for event occurrences have to be added.

[2]Differently from the original definition of the Event Calculus [11], devoid of any notion of event type and then only supporting an extensional definition of *initiates* and *terminates* predicates at the instance level, domain relations are intensionally defined in terms of event and property *types*.

```
broken_during(P, [Start,End]):-
  (terminates_at(E,P,T);initiates_at(E,P,T)),
  Start lt T, End gt T.
```

This axiom provides a so-called *strong interpretation* of *initiates_at* and *terminates_at* predicates: a given property P ceases to hold at some point T during the time-interval $[Start, End]$ if there is an event E which initiates or terminates P occurring at a time T belonging to $[Start, End]$. Notice that if we record three events $e1$, $e2$ and $e3$ such that $e1$ precedes $e2$, $e2$ precedes $e3$, both $e1$ and $e2$ initiates a property p and $e3$ terminates p, we can only conclude that p holds between the occurrence times of $e2$ and $e3$. This behaviour can be explained as follows: the strong interpretation assumes that an event terminating p actually occurred between $e1$ and $e2$, but it is not known when it occurred, and thus it is not possible to derive any validity interval for p between $e1$ and $e2$. According to this interpretation, pending events like $e1$ characterize situations of incomplete information about event occurrences. An alternative interpretation of *initiates_at* and *terminates_at* predicates, called *weak interpretation*, is also possible. According to such an interpretation an event initiates a property unless it has been already initiated and not yet terminated. To support this *weak interpretation* the definition of *broken_during* must be revised so that only terminating events can break the validity of property P [8].

Finally, we add the *holds_at* axiom relating a property to a time-point rather than to a time-interval:

```
holds_at(P,T):-
  mholds_for(P,[Start,End]), T gt Start, T le End.
```

The *holds-at* predicate conventionally assumes that a property is not valid at the starting point of the MVI, while it is valid at the ending point.

These axioms constitute the kernel of basic (typed) EC. They provide a simple and effective tool to reason about events and their effects, but have a limited expressive power. In particular, they provide no primitives for modeling relevant features such as context-dependency, discrete and continuous processes, time granularity. Several extensions [5, 7, 12, 13, 19, 18, 20] have been proposed to overcome these limitations. In this paper, we deal with the addition of context-dependency.

In basic EC, both *initiates_at* and *terminates_at* are *context-independent* predicates: the occurrence of an event of the given type initiates, or terminates, the validity of the relevant property *whatever is the context in which it occurs.* On the contrary, making *initiates_at* and *terminates_at* predicates context-dependent allows us to state that the occurrence of an event of a given type at a certain time point initiates or terminates the validity of the associated property *provided that some given conditions hold at such an instant* [3].

Formally, *initiates_at* and *terminates_at* predicates are generalized as follows[3]:

```
initiates_at(event,[prop1,...,propN],prop,T):-
  happens_at(event,T),
  holds_at(prop1,T),...,holds_at(propN,T).

terminates_at(event,[prop1,...,propN],prop,T):-
  happens_at(event,T),
  holds_at(prop1,T),...,holds_at(propN,T).
```

where N is greater than 0 when *initiates_at* and *terminates_at* are context-dependent and equal to 0 when they are context-independent. In this last case, we simply define *initiates_at* and *terminates_at* as:

```
initiates_at(E,[],P,T):-
  initiates_at(E,P,T).

terminates_at(E,[],P,T):-
  terminates_at(E,P,T).
```

As shown elsewhere, the inclusion of *holds_at* atoms in the body of *terminates_at* and *initiates_at* makes EC no more stratified [17, 4]. As a consequence, termination of the computation of MVIs is not guaranteed anymore.

2.2 Comparing EC with and without context-dependency

Consider a simple lighting system that is operated by its user using just one switch, whose functioning can be described as follows. The user can set the switch in one of two positions: on or off. If there is electrical power available, the effect of setting the switch in the on or off position is to switch the light on or off. If there is no electrical power available, the effect of setting the switch in the on position is delayed until electrical power is provided. A failure in providing power can anticipate the effect of setting the switch in the off position. We identify four types of events and three types of properties. Event types are: turnOn (the user sets the switch in the On position), turnOff (the user sets the switch in the Off position), pwrFail (a failure in the electrical power distribution network), pwrRstr (the failure is fixed, and power is restored). Property types are: switchOn (the position of the switch is on), pwrAvail (electrical power is available), lightsOn (the lights are lit). Using EC extended with context-dependency, the knowledge about effects of events on properties can be formalized as follows:

```
initiates_at(turnOn,[],switchOn,T):-
  happens_at(turnOn,T).

terminates_at(turnOff,[],switchOn,T):-
  happens_at(turnOff,T).

initiates_at(pwrRstr,[],pwrAvail,T):-
  happens_at(pwrRstr,T).
```

[3]The 2nd argument in the 4-argument version of *initiates_at* and *terminates_at* allows to statically inspect domain axioms in order to detect dependencies among properties as shown in Section 3.2.

```
terminates(pwrFail,[],pwrAvail,T):-
  happens_at(pwrFail,T).

initiates_at(turnOn,[pwrAvail],lightsOn,T):-
  happens_at(turnOn,T),
  holds_at(pwrAvail,T).

terminates_at(turnOff,[pwrAvail],lightsOn,T):-
  happens_at(turnOff,T),
  holds_at(pwrAvail,T).

initiates_at(pwrRstr,[switchOn],lightsOn,T):-
  happens_at(pwrRstr,T),
  holds_at(switchOn,T).

 terminates_at(pwrFail,[switchOn],lightsOn,T):-
  happens_at(pwrFail,T),
  holds_at(switchOn,T).
```

On the contrary, in the case of basic EC the modeler would be forced to use context-independent *initiates_at* and *terminates_at* predicates also to represent context-dependent knowledge. This can be done only by introducing additional types of events and naming them properly to highlight that they assume a given context. Considering for example the effect of the turnOn event on the property lightsOn, we can reformulate it with the EC axiom:

```
initiates_at(turnOnwithPwrAvail,lightsOn,T):-
  happens_at(turnOnwithPwrAvail,T).
```

This not only introduces a more complex event, but forces us to introduce another type of event (turnOnwithNoPwrAvail) in order to rewrite the description of the effects of turnOn on the property switchOn.

The drawbacks of this solution affect both the modeler and the final user of the database. On one hand, the modeler is indeed forced to (i) devise and introduce a number of new events that increases greatly with the complexity of the considered domain and the possible combinations of preconditions, (ii) highlight the differences among these events using long, awkward names. On the other hand, the final user is burdened with the responsability of precisely evaluating contexts in the real world in order to choose the proper event that has to be entered in the database (for example, in the simple lighting system example, the user must know if power was supplied when he/she pushed the button). Moreover, it becomes impossible in this way to enter an event when there's only partial knowledge about the context it assumes. These limitations become unbearable in complex domains and unnecessarily narrow the deductive power of the database. If EC is indeed extended with context-dependency, it becomes able to automatically identify contexts and therefore the user has only to enter basic types of events which he/she easily recognizes, leaving to the database the task of evaluating and updating the context and the effects. Furthermore, the database becomes able to reason with incomplete knowledge about contexts.

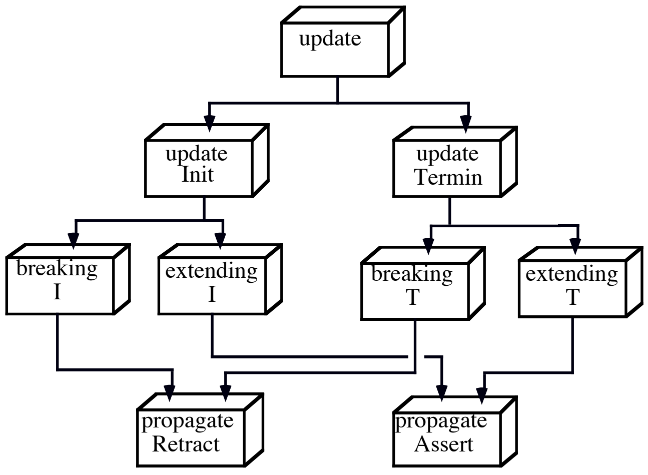

Figure 1: Architecture of CEC.

3 The Cached Event Calculus (CEC)

In this section we describe the main features of the Cached Event Calculus (CEC). It extends EC with a caching mechanism that receives instances of events as input and updates accordingly the cached set of MVIs (*mholds_for* assertions). Temporal reasoning in EC is performed at query time: the system logs any input event without processing it, and accesses the log when a query is posed. Thanks to the negation as failure rule, conclusions no longer supported are not derived anymore. The fact that the results of computations are never cached for later use makes query processing in EC very inefficient. A great deal of unnecessary computation is indeed performed whenever the same *mholds_for* query about a property p is processed several times between two consecutive updates affecting MVIs of p, because EC repeats each time the whole computation from the beginning instead of saving previous conclusions. Even worse, when context-dependency is added multiple computations of identical *mholds_for* (sub)queries can occur during the processing of a single *mholds_for* query, due to contexts evaluation. To eliminate these useless computations, EC should be provided with a mechanism for caching the currently believed set of MVIs, so that accesses to cached data replace blind recomputations in answering *mholds_for* queries. However, conceiving a suitable caching mechanism is not a trivial task, because it should be able to incrementally add new results to the old ones and to update or delete only the old results which need it. Moreover, differently from general caching mechanisms, we do not only need to add and/or remove assertions, but also to clip and/or extend existing MVIs according to domain and temporal knowledge.

The general architecture of CEC is depicted in Figure 1, where all its conceptual modules and their hierarchical relations are highlighted. We briefly summarize here the purpose of the modules; their functioning will be described in detail in the next sections. Each new event is entered into the database by *update*; *updateInit* and *updateTermin* are then called in order to manage properties which are initiated and terminated by the event, respectively. In both cases, the event can lead either to clip an existing MVI (this case is handled by *breakingI* and *breakingT*) or to possibly create a new MVI (this case is handled by *extendingI* and *extendingT*). When a MVI (or a part of it) is retracted (asserted), *propagateRetract* (*propagateAssert*) takes care of propagating that change to properties which depend on the changed one. Propagation of assertion and retractions can recursively activate the process of breaking or extending MVIs.

In the next two sections, we first describe the process of breaking and extending MVIs and then the process of propagating retractions and assertions.

3.1 Breaking and adding maximal validity intervals

CEC allows to enter events in the database by means of the *update* predicate. This predicate first explicitly records the instance of the event and then the properties initiated and terminated by the event are considered separately, by means of the *updateInit* and *updateTermin* predicates:

```
update(E,T):-
  assert(happens_at(E,T)),
  bagof(P,(updateInit(E,T,P);
           updateTermin(E,T,P)),_).

update(_,_).
```

If E initiates a property $Prop$ at time T, CEC tests if there exists a MVI $[T1,T2]$ for $Prop$ such that $T1 \leq T < T2$:

```
updateInit(E,T,Prop):-
  initiates_at(E,_,Prop,T),
  (insideLeftClosedInt(Prop,T,[T1,T2]) ->
     breakingI(Prop,T,[T1,T2]);
     extendingI(Prop,T)).

insideLeftClosedInt(Prop,T,[T1,T2]):-
    mholds_for(Prop,[T1,T2]), T1 le T, T lt T2.
```

If such an interval exists, we face two possibilities, which are handled by the *breakingI* predicate: (i) if $T1 = T$, there is already an event occuring at T, that initiates $Prop$ and then no changes are needed to interval $[T1,T2]$ at the moment; (ii) otherwise, interval $[T1,T2]$ is shortened in such a way that the new starting point becomes T, $Prop$ does not hold anymore in the clipped part $[T1,T]$, and the retraction of $[T1,T]$ has to be propagated.

```
breakingI(_,T1,[T1,_]):-!.

breakingI(Prop,T,[T1,T2]):-
  retract(mholds_for(Prop,[T1,T2])),
  assert(mholds_for(Prop,[T,T2])),
  propagateRetract([T1,T],Prop).
```

If instead there are no MVIs for $Prop$ satisfying the test $T1 \leq T < T2$, then T is the starting point of a new MVI for $Prop$. The new interval of validity starting at T is added by the *extendingI* predicate, and the new assertion has to be propagated.

```
extendingI(Prop,T):-
  new_termination(Prop,[T,NewEnd]),
  assert(mholds_for(Prop,[T,NewEnd])),
  propagateAssert([T,NewEnd],Prop).
```

The *new_termination* predicate finds the ending point of the new interval, distinguishing the following two cases: (i) there is a 'pending' terminating event for property $Prop$ occurring at time $NewEnd$ after T and thus the new MVI becomes $[T, NewEnd]$, (ii) there are no initiating or terminating events for property $Prop$ occurring after T and thus the new MVI becomes $[T, infPlus]$:

```
new_termination(Prop,[T,NewEnd]):-
  terminates_at(_,_,Prop,NewEnd),
  T lt NewEnd,
  \+broken_during(Prop,[T,NewEnd]), !.

new_termination(Prop,[T,infPlus]):-
  \+broken_during(Prop,[T,infPlus]).
```

The case in which the entered event E terminates instead of initiating a property is handled by a set of predicates (*updateTermin*, *insideRightClosedInt*, *breakingT*, *extendingT*, *new_initiation*), that is symmetrical to the previously discussed one [6].

3.2 Propagation of retractions and assertions

Each time a MVI $[T1,T2]$ for a property is retracted (asserted), the update has to be propagated to properties whose validity may rely on such an interval. The retraction (assertion) of $[T1,T2]$ indeed modifies the context of events occurring at time points belonging to it and, then, it can possibly invalidate (activate) their effects. More precisely, only those context-dependent *initiates_at* or *terminates_at* clauses having the retracted (asserted) property as a condition have to be reconsidered over the interval (we implement this selection by inspecting the domain axioms using the standard *clause* predicate).

In the case of propagation of retractions, we distinguish two relevant cases:

(i) possibly invalidated initiations
the retracted property $RetractedProp$ is a condition for the initiation of property P at the occurrence of event E and the occurrence time $T3$ of E belongs to $[T1,T2]$. In this case, the right part of the retracted interval $[T1,T2]$ overlaps the possibly affected MVI $[T3,T4]$ for P:

```
propagateRetract([T1,T2],RetractedProp):-
  clause(initiates_at(E,PropList,P,_),_),
  memberchk(RetractedProp,PropList),
  happens_at(E,T3),
  rightOverlap([T1,T2],P,[T3,T4]),
  retractForRightOverlap(P,T2,[T3,T4]),
  fail.

rightOverlap([T1,T2],P,[T3,T4]):-
  T1 lt T3, T3 le T2,
    mholds_for(P,[T3,T4]).
```

(ii) possibly invalidated terminations

RetractedProp is a condition for the termination of property P at the occurrence of event E and the occurrence time $T4$ of E belongs to $[T1, T2]$. In this case, the left part of the retracted interval $[T1, T2]$ overlaps the possibly affected MVI $[T3, T4]$ for P:

```
propagateRetract([T1,T2],RetractedProp):-
  clause(terminates_at(E,PropList,P,_),_),
  memberchk(RetractedProp,PropList),
  happens_at(E,T4),
  leftOverlap([T1,T2],P,[T3,T4]),
  retractForLeftOverlap(P,T1,[T3,T4]),
  fail.

leftOverlap([T1,T2],P,[T3,T4]):-
  T1 lt T4, T4 le T2,
  mholds_for(P,[T3,T4]).
```

The clauses for *propagateRetract* end with a fail predicate in order to cause the examination of all affected properties. When no more initiations and terminations are left to examine, a third clause guarantees success:

```
propagateRetract(_,_).
```

When an initiation is invalidated, there are four possible situations:

1. independency

   P still initiates at $T3$, because there exists a successful *initiates_at* clause, which does not include *RetractedProp* as condition:

   ```
   retractForRightOverlap(P,_,[T3,_]):-
     initiates_at(_,_,P,T3),!.
   ```

   The MVI for P is thus unchanged.

2. revised initiation with finite termination

   property P terminates at a time instant $T4$ (it does not hold forever) and either there exists an initiating event preceding $T4$ or no initiating or terminating events occur before $T4$ and P is assumed to hold from *infMin* by default. The *new_initiation* predicate is used to identify the proper case, determining the new starting point *NewStart* of the interval. Finally, if $T3 < NewStart$ then the retraction of validity over $[T3, NewStart]$ is propagated, otherwise the extension of validity over $[NewStart, T3]$ is propagated.

   ```
   retractForRightOverlap(P,_,[T3,T4]):-
    T4 \== infPlus,
    new_initiation(P,[NewStart,T4]), !,
    retract(mholds_for(P,[T3,T4])),
    assert(mholds_for(P,[NewStart,T4])),
    ((T3 lt NewStart) ->
         propagateRetract([T3,NewStart],P);
         propagateAssert([NewStart,T3],P)).
   ```

3. revised initiation with infinite termination

   property P holds until infPlus and there exists an event occurring at *NewStart* that initiates P with P holding uninterrupted after *NewStart*, that is, there are no events initiating or terminating P occurring after *NewStart*. In such a case *NewStart* becomes the new starting point and this modification is propagated: in case $T3 < NewStart$, the retraction of validity over $[T3, NewStart]$ is propagated; otherwise, the extension of validity over $[NewStart, T3]$ is propagated.

   ```
   retractForRightOverlap(P,_,[T3,infPlus]):-
    initiates_at(_,_,P,NewStart),
    \+broken_during(P,[NewStart,infPlus]),!,
    retract(mholds_for(P,[T3,infPlus])),
    assert(mholds_for(P,[NewStart,infPlus])),
    ((T3 lt NewStart) ->
        propagateRetract([T3,NewStart],P);
        propagateAssert([NewStart,T3],P)).
   ```

4. vanishing

   if none of the above described situations applies, the MVI for P is fully retracted and this retraction is propagated:

   ```
   retractForRightOverlap(P,_,[T3,T4]):-
     retract(mholds_for(P,[T3,T4])),
     propagateRetract([T3,T4],P).
   ```

In the case of invalidated terminations, there are four possible situations, which are handled by *retractForLeftOverlap* that is symmetrical to the previously described predicate [6].

In the case of propagation of assertions, we distinguish two relevant cases:

(i) possibly new initiations

the asserted property *AssertedProp* is a condition for the initiation of property P at the occurrence of event E, the occurrence time T of E belongs to $[T1, T2]$ and there is not already a MVI for P with T as its starting point. In this case, the already described *updateInit* predicate is used to check if P is now initiated at T and possibly revising the database accordingly.

```
propagateAssert([T1,T2],AssertedProp):-
  clause(initiates_at(E,PropList,P,_),_),
  memberchk(AssertedProp,PropList),
  happens_at(E,T),
  T1 lt T, T le T2,
  \+mholds_for(P,[T,_]),
  updateInit(E,T,P),
  fail.
```

(ii) possibly new terminations

the asserted property *AssertedProp* is a condition for the termination of property P at the occurrence of event E, the occurrence time T of E belongs to $[T1, T2]$ and there is not already a MVI for P with T as its ending point. In this case, the already described *updateTermin* predicate is used to check if P is now terminated at T and possibly revising the database accordingly.

```
propagateAssert([T1,T2],AssertedProp):-
  clause(terminates_at(E,PropList,P,_),_),
  memberchk(AssertedProp,PropList),
  happens_at(E,T),
```

```
        T1 lt T, T le T2,
        \+mholds_for(P,[_,T]),
        updateTermin(E,T,P),
        fail.
```

As in the case of *propagateRetract*, the fail predicate has been used to force backtracking in order to examine all affected properties. Therefore, a third clause guarantees success as before:

```
propagateAssert(_,_).
```

3.3 Running the lighting system example in CEC

We will now show how CEC builds and maintains the set of cached MVIs, by examining one execution of the lighting system example. Suppose to perform the following updates in the shown order, that is not chronological:

```
  update(initially(pwrAvail),0),
  update(turnOff,8),
  update(turnOn,4),
  update(turnOn,10),
  update(pwrFail,6),
  update(pwrRstr,12).
```

Figure 2 illustrates pictorially the effects of each of the six updates. The effect of the first update is to initiate a pwrAvail property that holds between 0 and infPlus (extendingI with initial conditions). Propagation of the assertion of this property has no effect because there are no influenced events in the database. The effects of the second update are to terminate: (i) a lightsOn interval that holds between infMin and 8 and (ii) a switchOn interval that also holds between infMin and 8 (extendingT in both cases). Propagation of assertion for both properties has no effect because there are no other events in the interval of time considered by propagation. The effects of the third update are: (i) the MVI of property lightsOn is broken at 4, and validity between infMin and 4 is thus retracted and (ii) the MVI of property switchOn is also broken at 4, and validity between infMin and 4 is thus retracted (brokenI in both cases). Propagation of retraction for both properties has no effect because there are no other events in the interval of time considered by propagation. The effects of the fourth update are to initiate: (i) a lightsOn interval that holds between 10 and infPlus and (ii) a switchOn interval that also holds between 10 and infPlus (extendingI in both cases). Propagation of assertion for both properties has no effect because there are no other events in the interval of time considered by propagation. The effects of the fifth update are: (i) the MVI of property pwrAvail is broken at 6 and validity between 6 and infPlus is thus retracted (brokenT), (ii) retraction is propagated and leads to reconsider the effects of event turnOn at 10, leading to the retraction of property LightsOn holding between 10 and infPlus (vanishing), (iii) the effects of event turnOff at 8 are then reconsidered and property LightsOn is retracted between 6 and 8 (revised termination with finite initiation). Propagation of the retractions of lightsOn have no effect because there are no influenced events. The effects of the sixth update are to initiate: (i) a pwrAvail interval that holds between 12 and infPlus and (ii) a lightsOn interval that also holds between 12 and infPlus (extendingI in both cases). Propagation of assertion for both properties has no effect because there are no other events in the interval of time considered by propagation.

As already pointed out, changing the order of execution of the six updates has no influence on the final contents of the database, but could affect efficiency. In particular, if the complete sequence of events was entered chronologically, there would be no need for a substantial revision of the database as that caused by the fifth update.

4 Complexity Analysis

In this section we analyze the complexity of executing EC and CEC with an ordinary Prolog intepreter. We focus on the execution of *mholds_for* queries returning the full set of MVIs for a given property *prop*. Such queries take the following form:

```
  ?- bagof(MVI,mholds_for(prop,MVI),MVIs).
```

We also assume that the database contains a set of n initiating events and n terminating events for any property. First of all, we determine the complexity of query processing in EC devoid of context-dependency, and show how EC performance heavily decreases when context-dependency is added. Then, we prove how the addition of a caching mechanism strongly reduces the cost of query processing. We show that the cost of query processing in CEC, with and without context-dependency, is *linear*, and that the complexity of CEC update processing is less than the complexity of EC query processing except in the case of context-independency where it is equal. In all proofs we assume the strong interpretation of *initiates_at* and *terminates_at* predicates. It is possible to show that in the case of weak interpretation the worst-case analysis leads to the same results. We only sketch out the structure of proofs; the details can be found in [6].

4.1 The complexity of query processing in EC

We initially consider the case of a database with only context-independent definitions of *initiates_at* and *terminates_at*. It is possible to prove the following theorem.

Theorem 4.1
The complexity of EC query processing, measured in terms of accesses to happens_at facts, is $\mathcal{O}(n^3)$, where n is the number of initiating (terminating) events in the database for the considered property.

We performed the proof in two steps. We first determined a cubic upper bound ($2{\cdot}n^3{+}n^2{+}n$) for the total number of accesses to *happens_at* facts in answering

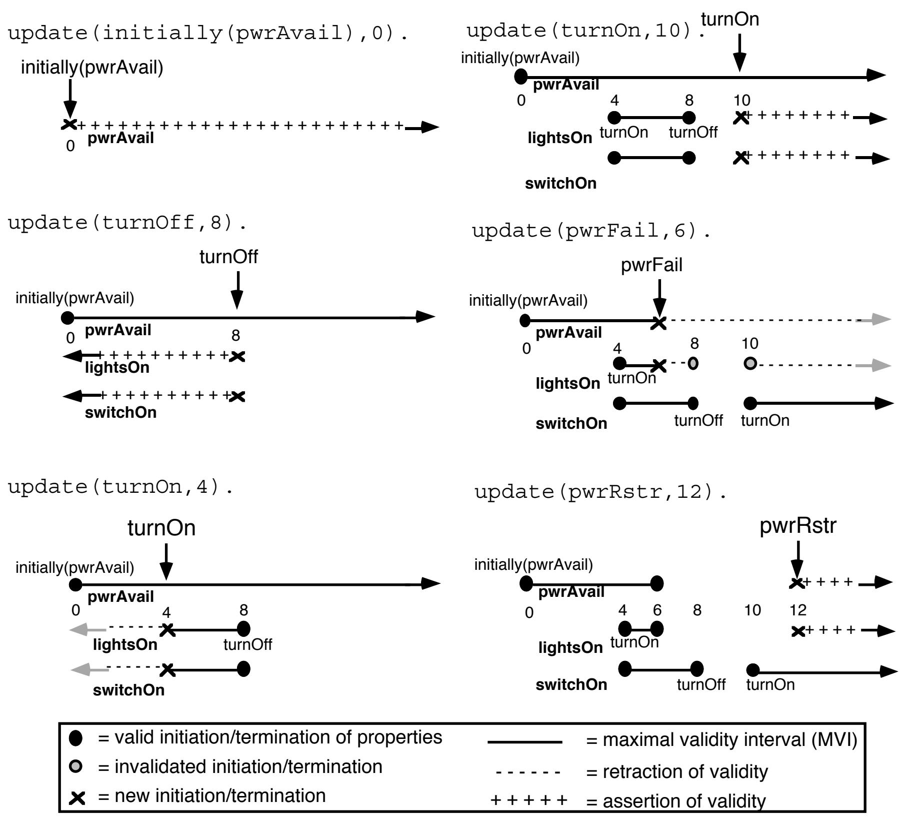
update(initially(pwrAvail),0).
initially(pwrAvail)
pwrAvail
update(turnOn,10).
turnOn
lightsOn
switchOn
turnOff
update(turnOff,8).
update(pwrFail,6).
pwrFail
update(turnOn,4).
update(pwrRstr,12).
pwrRstr
● = valid initiation/termination of properties
○ = invalidated initiation/termination
× = new initiation/termination
= maximal validity interval (MVI)
- - - - - - = retraction of validity
+ + + + + = assertion of validity

Figure 2: A sample execution of CEC.

mholds_for queries with only the property argument instantiated. Then we showed that the cubic limit is actually reached.

In order to determine the complexity of query processing in EC with context-dependency some preliminary notions are necessary. We must take into account that a condition of a context-dependent *initiates_at* or *terminates_at* may itself be context-dependent. Therefore, we must consider in general an arbitrary nesting level of context-dependent properties. Let L_{bk} be the maximum level of nesting from a single property. To formally define L_{bk}, we introduce the notion of property dependency graph associated with a set of *initiates_at* and *terminates_at* clauses. A *property dependency graph* is a directed acyclic graph where[4]:

(i) each vertex denotes a property p;

(ii) there exists an edge (p_j, p_i) if and only if there exists an *initiates_at* clause for p_i having p_j as one of its conditions or a *terminates_at* clause for p_i having p_j as one of its conditions.

L_{bk} is defined as the length of the longest path in the graph. It is possible to prove the following theorem by induction on the nesting level of context-dependent properties L_{bk}.

Theorem 4.2
The complexity Comp(query(L_{bk})) of EC query processing, measured in terms of accesses to happens_at facts, is $\mathcal{O}(n^{(L_{bk}+1)\cdot 3})$, where n is the number of initiating (terminating) events in the database for the considered property.

If we indeed consider $L_{bk} = 0$, we fall in the already discussed case of the basic calculus, whose complexity has been shown to be $\mathcal{O}(n^3)$. What happens when L_{bk} is 1 or more is that the evaluation of each condition in the *initiates_at* or *terminates_at* predicates for p will result in the evaluation of a *mholds_for* predicate for this condition with the temporal argument unbound and a $L_{bk} - 1$ nesting level. We showed that the relationship between the complexity upper bounds for $query(L_{bk})$ and $query(L_{bk} - 1)$ is expressed by the recurrent expression:

$$Comp(query(L_{bk})) = n^2 \cdot 2n \cdot (1 + C_{bk} \cdot Comp(query(L_{bk} - 1)))$$

[4]The requirement that the graph is acyclic is needed to make it possible the comparison between EC and CEC complexities. EC indeed loops whenever there is a cycle among context-dependent properties. On the contrary, there exist sufficient conditions for the termination of CEC also in case of property cycles (the main condition is that at each time instant at most one of the properties involved in the property cycle may hold). Examples of non-critical property cycles in a medical application domain are given in [8]. The general problem of looping in EC and CEC is currently under investigation.

were C_{bk} is the maximum number of conditions found in the context-dependent *initiates_at* or *terminates_at* clauses. We then proved by induction that the order of complexity of $query(L_{bk})$ is at most $\mathcal{O}(n^{(L_{bk}+1)\cdot 3})$. As in the basic calculus, it is straightforward to show that this limit is actually reached[5].

4.2 The complexity of update processing in CEC

In the case of CEC, the complexity has to be measured in terms of accesses to both *happens_at* and *mholds_for* facts. Differently from EC, where each evaluation of a *mholds_for* query results into a number of accesses to *happens_at* facts, in CEC *mholds_for* predicates are explicitly recorded in the database as facts. Under the given assumption that there is a set of n initiating events and n terminating events for every property in the database, there are at most n disjoint MVIs for each property (n *mholds_for* facts recorded in the database). Thus all EC accesses to *happens_at* facts for a *mholds_for* query for a given property collapse into at most n CEC accesses to *mholds_for* facts about such a property. Therefore, the cost of a *mholds_for* query in CEC is *linear* in the number of cached MVIs for the considered property, whatever is the value of L_{bk}.

Differently from EC, the complexity of update processing in CEC is not constant at all. To precisely determine such a complexity some preliminary notions are needed. First of all, let P be the maximum number of properties initiated or terminated by a single event. Furthermore, we must take into account that the assertion of a new *mholds_for* fact (the retraction of an existing *mholds_for* fact) may cause (suppress) the initiation or the termination of a property depending on it and then the assertion of an additional *mholds_for* fact (the retraction of an existing *mholds_for* fact) concerning such a property. Therefore, we must consider in general an arbitrary level of propagation of assertions (retractions). Let L_{fw} be the maximum level of propagation from a single property. L_{fw} can be formally defined as the length of the longest path in the property dependency graph, and then it it is equal to L_{bk}[6].

It is possible to prove the following lemma by induction on the propagation level L_{fw}.

Lemma 4.3
The complexity of propagating assertions, measured in terms of accesses to happens_at and mholds_for facts, is $\mathcal{O}(n^{L_{fw}+3})$, where n is the number of initiating (terminating) events in the database for the considered property, and it is equal to the complexity of propagating retractions.

The proof is accomplished in three steps. We first determine the recurrent expression of the costs of *propagateRetract* and *propagateAssert* with respect to a level of propagation L_{fw} in terms of their costs with respect to the level $L_{fw}-1$; then we prove that the cost of *propagateRetract* is always less than the cost of *propagateAssert*; finally we provide the general cost expressions of *propagateRetract* and *propagateAssert*.

On the basis of *Lemma 4.3*, it is straightforward to prove the following theorem about the complexity of update processing in CEC.

Theorem 4.4
The complexity $Comp(update(L_{fw}))$ of CEC update processing, measured in terms of accesses to happens_at and mholds_for facts, is $\mathcal{O}(n^{L_{fw}+3})$, where n is the number of initiating (terminating) events in the database for the considered property.

It is worth noting that in the context-independent case ($L_{fw}=0$), the complexity of update processing is $\mathcal{O}(n^3)$. In such a case there are neither propagation of assertions nor propagation of retractions, and the worst-case complexity of update processing is just the complexity of adding at most P new *mholds_for* facts. The orders of complexity of query and update processing in EC and CEC are summarized in Table 1.

5 Conclusions

This paper has proposed a caching mechanism for an efficient implementation of Kowalski and Sergot's Event Calculus (EC). In the context-independent case ($L_{bk} = L_{fw} = 0$), the cached version of EC we developed (CEC) makes the complexity of query processing *linear*, shifting temporal reasoning from query to update processing. The complexity of update processing in CEC is indeed equal to the complexity of query processing in EC. In the more significant context-dependent case ($L_{bk} = L_{fw} \geq 1$), CEC allows us to obtain an absolute improvement in performance because the order of complexity of update processing in CEC ($L_{fw}+3$) is strictly lower than the order of complexity of query processing in EC ($(L_{bk}+1)\cdot 3$). CEC has been fully implemented on a Sun Sparc2 in Quintus Prolog and extensively tested in order to ensure that it yields the same outputs as EC (but much more efficiently).

Acknowledgements

We would like to thank one of the anonymous referees for the useful and perceptive comments. This work has been partially supported by the Italian National Research Council (Special Project on Management of Temporal Information in Data and Knowledge Bases), and the P.A.O.L.A. Consortium, whose members are ASEM Resolution, INSIEL, and Università di Udine.

[5]It is worth noting that this is a worst case analysis. In fact, C_{bk} and L_{bk} can be redefined for each single property to evaluate the complexity of specific queries or classes of queries.

[6]Note that if L_{bk} and L_{fw} are redefined to account for nesting and propagation levels of specific properties, they are not necessarily equal anymore.

	EC update	EC query	CEC update	CEC query
$L_{bk} = L_{fw} = 0$	*const*	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$
$L_{bk} = L_{fw} = 1$	*const*	$\mathcal{O}(n^6)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n)$
$L_{bk} = L_{fw} = 2$	*const*	$\mathcal{O}(n^9)$	$\mathcal{O}(n^5)$	$\mathcal{O}(n)$
...	...	...	...	...
$L_{bk} = L_{fw} = k$	*const*	$\mathcal{O}(n^{(k+1)\cdot 3})$	$\mathcal{O}(n^{k+3})$	$\mathcal{O}(n)$

Table 1: Comparing complexities of EC and CEC.

References

[1] J. Allen, *Planning as Temporal Reasoning* in Proc. of KR-91, Cambridge, MA, pp.3–14.

[2] M. Baudinet, J. Chomicki, P. Wolper, *Temporal Deductive Databases. Chapter 13* in [21], pp. 294–320.

[3] M. Boddy, *Temporal Reasoning for Planning and Scheduling*, SIGART Bulletin, Vol.4, No.3, 1993, pp.17-20.

[4] I. Cervesato, A. Montanari, A. Provetti, *On the Non-monotonic Behavior of Event Calculus for Deriving Maximal Time-Intervals*; to appear in The International Journal of Interval Computations, 1994.

[5] L. Chittaro, A. Montanari, *Experimenting a Temporal Logic for Executable Specifications in an Engineering Domain*, in G. Rzevski, J. Pastor, R.A. Adey (eds), *Applications of Artificial Intelligence in Engineering VIII*, Computational Mechanics Publications & Elsevier Applied Science, Boston and London, 1993, pp. 185–202.

[6] L. Chittaro, A. Montanari, *Facing Efficiency and Looping Problems of the Event Calculus through Caching. Part I: Efficiency*; Research Report RR18/93, Dipartimento di Matematica e Informatica, Università di Udine, November 1993.

[7] L. Chittaro, A. Montanari, A. Provetti, *Skeptical and Credulous Event Calculi for Supporting Modal Queries*; to appear in Proc. of ECAI'94, Amsterdam, The Netherlands, Wiley & Sons Publishers, August 1994.

[8] L. Chittaro, A. Montanari, M. Dojat, C. Gasparini, *The Event Calculus at work: a Case Study in the Medical Domain*; (submitted), 1994.

[9] T. Dean, D. McDermott, *Temporal Data Base Management*; Artificial Intelligence, Vol. 32, 1987, pp. 1–55.

[10] T. Dean, *Using Temporal Hierarchies to Efficiently Mantain Large Temporal Databases*; Journal of the ACM, Vol. 36, No. 4, October 1989, pp. 687–718.

[11] R. Kowalski, M. Sergot, *A Logic-based Calculus of Events*; New Generation Computing, 4, 1986, pp. 67–95.

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

[13] A. Montanari, E. Maim, E. Ciapessoni, E. Ratto, *Dealing with Time Granularity in the Event Calculus*; Proceedings FGCS-92, Fifth Generation Computer Systems, Tokyo, Japan, IOS Press, 1992, pp. 702–712.

[14] A. Montanari, B. Pernici, *Temporal Reasoning. Chapter 21* in [21], pp. 534–562.

[15] J. Pinto, R. Reiter, *Temporal Reasoning in Logic Programming: A Case for the Situation Calculus*, Proc. ICLP'93, Budapest, Hungary, 1993, pp. 203-221.

[16] R. Reiter, *Proving Properties of States in the Situation Calculus*; Artificial Intelligence, Vol. 64, no.2, 1993, pp. 337-351.

[17] M. Shanahan, *Prediction is Deduction, but Explanation is Abduction*; Proc. of IJCAI'89, Detroit, The MIT Press, 1989, pp. 1055–1060.

[18] M. Shanahan, *Representing Continuous Change in the Event Calculus*; Proc. of ECAI'90, Stockholm, Sweden, 1990, pp. 598–603.

[19] M. Sergot, *(Some Topics in) Logic Programming in AI*; Advanced School on Foundations of Logic Programming, Alghero, Italy, 1990.

[20] S. Sripada, *Temporal Reasoning in Deductive Databases*; PhD thesis in Computing, Imperial College, London, 1990.

[21] A. Tansell, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.), *Temporal Databases: Theory, Design and Implementation*, The Benjamin/Cummings Series on Database Systems and Applications, Benjamin/Cummings Publishers, 1993, pp. 534–562.