EFFICIENT TEMPORAL REASONING IN THE CACHED EVENT CALCULUS

L. CHITTARO AND A. MONTANARI

Dipartimento di Matematica e Informatica, Università di Udine, Udine 33100, Italy

This article deals with the problem of providing Kowalski and Sergot’s event calculus, extended with context dependency, with an efficient implementation in a logic programming framework. Despite a widespread recognition that a positive solution to efficiency issues is necessary to guarantee the computational feasibility of existing approaches to temporal reasoning, the problem of analyzing the complexity of temporal reasoning programs has been largely overlooked. This article provides a mathematical analysis of the efficiency of query and update processing in the event calculus and defines a cached version of the calculus that (i) moves computational complexity from query to update processing and (ii) features an absolute improvement of performance, because query processing in the event calculus costs much more than update processing in the proposed cached version.

Key words: event calculus, temporal reasoning complexity, database query and update, caching.

1. INTRODUCTION

Temporal reasoning is emerging as a major requirement in many intelligent system applications such as planning and scheduling (Boddy 1993). Considerable research efforts have been devoted to providing temporal reasoning tasks with a common formal basis, so that they can be defined in a uniform framework [for an up-to-date survey see Montanari and Pernici (1993)]. In contrast, efficiency issues involved in temporal reasoning have often been ignored, and this has been recognized as a major limiting factor to the application of temporal reasoning in realistic domains (Dean 1989).

In this article we deal with the problem of providing Kowalski and Sergot’s (1986) event calculus (EC) extended with context dependency, with an efficient implementation in a logic programming framework. From a description of events that occur in the real world, EC allows one to derive domain properties and the time periods during 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 of events’ occurrences. EC database and rules are formulated in a logic programming framework as a temporal deductive database. Unlike conventional databases, which are designed to store the most recent available data, temporal databases (Tansel et al. 1993) maintain past and present (and possibly future) data in a uniform framework. Temporal deductive databases (TDDs) give 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 (Sripada 1990; Baudinet et al. 1993). In the particular case of incomplete temporal data, they allow one to infer information that is neither explicitly asserted nor monotonically implied by the available knowledge by drawing conclusions according to suitable assumptions such as closed world or default persistence. Conclusions derived in this way are obviously defeasible and must be withdrawn as soon as the addition of further information makes them inconsistent.

Different TDDs have been proposed in the literature, e.g., Kowalski and Sergot (1986), Dean and McDermott (1987), Reiter (1992). They differ from each other either in the underlying model of change (state based vs event based) or in the programming language paradigm they adopt (functional vs logic programming), or in both. EC presents several advantages over relational temporal databases as well as over other TDDs (Kowalski 1992).

1Part of this work was carried out while the second author was visiting ILLC, University of Amsterdam.

© 1996 Blackwell Publishers, 238 Main Street, Cambridge, MA 02142, USA, and 108 Cowley Road, Oxford, OX4 1JF, UK.
First, while relational temporal databases only record the start and end of properties, thus losing the semantic structure of causing events, EC supports an explicit representation of events. In this way, updates are performed by entering events and deriving the start and end of properties as a logical consequence from event descriptions by means of general rules which express the semantics of events. Second, default persistence is defined both in the future and in the past. Third, insertion of events in the database is not required to follow the chronological order of their occurrences.

The basic issue in defining TDD’s temporal reasoning strategies concerns the relation between elaboration of input data to infer whatever can be inferred (update processing) and elaboration of submitted queries in order to compute their answers by accessing and reasoning on available data (query processing) (Montanari and Pernici 1993). If temporal reasoning is performed at update time, it has to infer all relevant consequences of input data when they are entered. In such a way, query processing is extremely simplified. If temporal reasoning is performed at query time, the system logs any input at update time without processing it and accesses and reasons about the log when a query is submitted. This is the case of EC: update processing consists in the addition of a new time-stamped event occurrence (other types of update are taken into consideration in Cervesato et al. (1993)), and its cost is constant. Temporal reasoning in EC is performed entirely at query time, when the effects of the logged events on the validity of properties are considered. EC computes the set of maximal validity intervals (MVIs) for a given property according to an expensive generate-and-test strategy. Moreover, since it does not record the MVIs it computes, it is not able to circumscribe the effects of updates in such a way that only MVIs which may have been affected by an update need to be reconsidered. The addition of context dependency heavily deteriorates performance, because it may happen that the same MVIs must be evaluated several times during the processing of a single query.

This article presents a cached event calculus (CEC) that extends EC with a MVIs generation and storage mechanism. CEC caches MVIs of properties for later use in query processing and possibly updates them when a new event is entered in the database. We show that the cost of querying CEC database about the set of MVIs for a given property is linear in the number of initiating and terminating events for that property recorded in the database and that it does not change when we add context dependency. Moreover, we prove that the cost of update processing in CEC is much less than the cost of query processing in EC.

The article is organized as follows. Section 2 introduces EC extended with context dependency. CEC is presented in detail in Section 3, which illustrates how entering an event in the database may cause the assertion of new MVIs for the properties it affects or the clipping of cached ones and how new assertions and retractions are possibly propagated to other properties. Section 4 analyzes and compares the complexity of query and update processing in EC and CEC. Conclusions provide an assessment of the achieved results and discuss current work and open issues.

2. THE EVENT CALCULUS WITH CONTEXT DEPENDENCY

Kowalski and Sergot’s event calculus is a general approach to representing and reasoning about events and their effects in a logic programming framework. Even if extensions of EC for dealing with hypothetical sequences of events have been defined in the literature, e.g., Eshghi (1988), EC was originally proposed to reason about actual courses of events (narratives). EC database updates provide information about the happening of events and their occurrence times and are of additive nature only. Moreover, whatever the state of the database at a given time, information about occurred events can possibly be incomplete, but not inconsistent.
From a description of events which occur in the real world and properties they initiate or terminate, EC derives the MVIs over which properties hold. 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. It embodies a notion of default persistence according to which properties are assumed to persist until an event occurs that interrupts them. Formally, we represent an event occurrence by means of the `happens_at(event, timepoint)` unit clause.\(^2\) The relation between events and properties is defined by means of `initiates_at` and `terminates_at` clauses:

\[
\text{initiates_at(event1,prop1,T)};\quad \text{terminates_at(event2,prop2,T)};\quad \text{happens_at(event1,T)};\quad \text{happens_at(event2,T)}.
\]

The `initiates_at` (`terminates_at`) clause states that each event of type `event1` (`event2`) initiates (terminates) a period of time during which property `prop1` (`prop2`) holds, respectively. A particular `initiates_at` clause is 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 (time point 0) can be derived by means of the clause:

\[
\text{initiates_at(initially(prop),prop,0)};\quad \text{happens_at(initially(prop),0)}.
\]

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

\[
\text{mholds_for(P,\{Start,End\})}:\quad (1.1)
\text{initiates_at(Ei,P,Start), terminates_at(Et,P,End), End \text{ gt Start}, \neg broken\_during(P,\{Start,End\}).}
\]

\[
\text{mholds_for(P,\{Start,\text{infPlus}\})}:\quad (1.2)
\text{initiates_at(Ei,P,Start), \neg broken\_during(P,\{Start,\text{infPlus}\}).}
\]

\[
\text{mholds_for(P,\{\text{infMin},End\})}:\quad (1.3)
\text{terminates_at(Et,P,End), \neg broken\_during(P,\{\text{infMin},End\}).}
\]

\[
\text{broken\_during(P,\{Start,End\})}:\quad (1.4)
\text{\{terminates_at(E,P,T);initiates_at(E,P,T)\}, Start \text{ lt T, End \text{ gt T}.}
\]

where the predicate `gt` extends the ordinary ordering relationship `>` to include the cases involving infinite arguments, syntactically denoted by `inft` and `infPlus`. Analogously, we defined the predicates `ge`, `lt`, and `le` which extend `\geq`, `<`, and `\leq`, respectively.

Clause (1.1) 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 `broken\_during` predicate is indeed interpreted using negation-as-failure. Clauses (1.2) and (1.3) deal with cases of persistence in the future and in the past, respectively. Clause (1.4) states that a given property `P` is interrupted during the time interval `[Start,End]` if there is an event `E` that happens between them and initiates or terminates `P`. This axiom provides the so-called strong interpretation of the `initiates_at` predicate: 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 time `T`. For example, consider an ordered sequence of events `e_1, \ldots, e_n`, each one initiating the same property `p`, followed by

\(^2\)The requirement of knowing exactly the occurrence time of each event can actually be relaxed. We dealt with the case of partially ordered events in Chittaro et al. (1994, 1995b) and Cervesato et al. (1995).
an event $e_{n+1}$ terminating $p$. According to a strong interpretation, EC derives the validity of $p$ only between the innermost initiating event $e_i$ and the terminating event $e_{n+1}$.

The strong interpretation is needed when dealing with incomplete sequences of events. For example, consider a switch that can take two different positions (on and off). Its behavior can be described by means of two events: one that changes the position from off to on, the other from on to off. While two “turn on” events cannot occur consecutively in the real world, it may happen that an incomplete sequence consisting of two consecutive turn on events, followed by a “turn off” event, is recorded in the database. A strong interpretation of $\text{initiates.at}$ allows EC to recognize that a missing turn off event must have occurred between the two turn on events. However, since it is not able to temporally locate such an event, it only concludes that the switch is on between the second turn on event and the turn off event, and considers the first turn on event as a pending initiating event.

An alternative interpretation of the $\text{initiates.at}$ predicate, called weak interpretation, is also possible. According to such an interpretation, a property $P$ is initiated by an initiating event unless it has already been initiated and not yet terminated. Consider again the ordered sequence of initiating events $e_1, \ldots, e_n$, followed by the terminating event $e_{n+1}$. According to the weak interpretation, EC would derive the validity of $p$ between the outermost initiating event $e_1$ and $e_{n+1}$. To support a weak interpretation the definition of $\text{broken.during}$ must be slightly revised so that only terminating events can break the validity of property $P$, and clause (1.3) must be omitted (under the weak interpretation, the presence of a MVI starting at $\infMin$ would make ineffective any occurrence of an initiating event for $P$ inside that MVI).

The weak interpretation is needed to aggregate homogeneous states. Consider the problem of monitoring patients who receive a partial mechanical respiratory assistance (Chittaro et al. 1995a). In this context, it often happens that data acquired with two consecutive samplings do not cause a transition in the classification of the patient ventilatory state. Adopting the weak interpretation, the second data acquisition does not clip the MVI of patient state initiated by the first one.

Finally, we add the $\text{holds.at}$ axiom relating a property to a time point rather than to a time interval:

$$\text{holds.at}(P,T) :- \text{holds-for}(P,[\text{Start,End}]), T \geq \text{Start}, T \leq \text{End}. \quad (1.5)$$

The $\text{holds.at}$ predicate conventionally assumes that a property is not valid at the starting point of the MVI, while it is valid at its ending point.

In EC, both $\text{initiates.at}$ and $\text{terminates.at}$ are context-independent predicates: The occurrence of an event of the given type initiates, or terminates, the validity of the relevant property $\text{whatever the context in which it occurs}$. To model preconditions, we add context-dependent $\text{initiates.at}$ and $\text{terminates.at}$ predicates, using the $\text{holds.at}$ predicate as in Shanahan (1989).

They allow 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 $\text{provided that some given conditions hold at such a time point}$.

Formally, $\text{initiates.at}$ and $\text{terminates.at}$ predicates are generalized as follows:

$$\text{initiates.at(event, [prop1, ..., propN], prop, T)} :- \text{happens.at(event, T)}, \text{holds.at(prop1, T), ..., holds.at(propN, T)}.$$

$$\text{terminates.at(event, [prop1, ..., propN], prop, T)} :- \text{happens.at(event, T)}, \text{holds.at(prop1, T), ..., holds.at(propN, T)}.$$

where $N$ is greater than 0 when $\text{initiates.at}$ and $\text{terminates.at}$ are context dependent and equal
to 0 when they are context independent. In this last case, we simply define \texttt{initiates\_at} and \texttt{terminates\_at} as:

\begin{verbatim}
initiates\_at(E,\{\},P,T):-
terminates\_at(E,\{\},P,T):-
initiates\_at(E,P,T).
terminates\_at(E,P,T).
\end{verbatim}

The second argument in the proposed four-argument version of \texttt{initiates\_at} and \texttt{terminates\_at} allows us to statically inspect domain axioms to detect dependencies among properties.

In general, preconditions can also take negative form. For simplicity, we deal here with positive conditions and translate negative conditions in positive form (e.g., \textit{not lightOn} becomes \textit{lightOff}). The explicit handling of conditions in negative form would not entail any floundering problem, because the \texttt{hold\_at} predicate would always be called with all its arguments instantiated. Moreover, it is possible to show that it would not change the worst-case complexity of both EC and its cached version. It would instead require splitting the second argument of the four-argument version of \texttt{initiates\_at} and \texttt{terminates\_at} to have both positive and negative condition lists, to be handled symmetrically in the processing activity described in the following.

As shown in Shanahan (1989), the inclusion of \texttt{hold\_at} atoms in the body of \texttt{terminates\_at} and \texttt{initiates\_at} makes EC no more stratified. As a consequence, in the general case termination of the computation of MVIs is not guaranteed anymore. However, it can be easily recovered by preventing the set of dependencies among properties from including loops.

As an example of a domain involving context-dependent \texttt{initiates\_at} and \texttt{terminates\_at} clauses, consider a simple lighting system that is operated by its user with 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 prevents the production of light until power is restored again. We identify four types of events and three types of properties. Event types are: \textit{turnOn} (the user sets the switch in the on position), \textit{turnOff} (the user sets the switch in the off position), \textit{pwrFail} (a failure in the electrical power distribution network), and \textit{pwrRstr} (the failure is fixed, and power is restored). Property types are: \textit{switchOn} (the position of the switch is on), \textit{pwrAvail} (electrical power is available), \textit{lightsOn} (the lights are lit). Using EC extended with context dependency, the knowledge about effects of events on properties can be formalized as follows:

\begin{verbatim}
initiates\_at(turnOn,\{\},switchOn,T):-
happens\_at(turnOn,T).
initiates\_at(pwrRstr,\{\},pwrAvail,T):-
happens\_at(pwrRstr,T).
initiates\_at(turnOn,\{pwrAvail\},lightsOn,T):-
happens\_at(turnOn,T), holds\_at(pwrAvail,T).
initiates\_at(pwrRstr,\{\},pwrAvail,T):-
happens\_at(pwrRstr,T), holds\_at(pwrAvail,T).
initiates\_at(pwrRstr,\{pwrAvail\},lightsOn,T):-
happens\_at(pwrRstr,T), holds\_at(pwrAvail,T).
initiates\_at(pwrFail,\{\},pwrAvail,T):-
happens\_at(pwrFail,T), holds\_at(pwrAvail,T).
initiates\_at(pwrRstr,\{pwrAvail\},lightsOn,T):-
happens\_at(pwrRstr,T), holds\_at(pwrAvail,T).
\end{verbatim}

This case study is used in the next section to exemplify the behavior of CEC.
3. THE CACHED EVENT CALCULUS

In this section we describe the main features of CEC. CEC extends EC with a caching mechanism that receives event occurrences as input and updates accordingly the cached set of MVIs (\textit{mholds} for assertions).

To determine the set of MVIs for a given property \(p\), clause (1.1) of EC generates all ordered pairs of initiating and terminating events for \(p\), and then for each pair checks the absence of known interrupting events in between. Clause (1.2) and (1.3) check for the absence of known interrupting events between each initiating event for \(p\) and \textit{infPlus}, and each terminating event for \(p\) and \textit{infMin}, respectively. Thanks to the negation-as-failure rule, MVIs no longer supported after a database update are not derived anymore. This generate-and-test strategy is the major source of EC inefficiency. Moreover, a great deal of unnecessary computation is performed whenever the same \textit{mholds} for query about a property \(p\) is processed several times between two consecutive updates affecting MVIs for \(p\), because EC repeats each time the whole computation from the beginning. Finally, when context dependency is added, multiple computations of identical \textit{mholds} for (sub)queries can occur during the processing of a single \textit{mholds} for query, due to contexts evaluation.

These efficiency problems are overcome by CEC. Entering a new event in the database generally affects a small number of MVIs, while most of them remain unchanged, and CEC is able to incrementally add new results to the old ones and to update or delete only the old results that need it. Unlike general caching mechanisms, e.g., Warren (1992), CEC takes into account domain and temporal knowledge about MVIs to optimize their addition and deletion. CEC does not only add and/or remove assertions, but also clips existing MVIs according to domain and temporal knowledge.

CEC replaces the generate-and-test strategy of EC with a more focused one: when the occurrence of an event of type \(e\) at time \(t\) is recorded in the database, CEC restricts update processing to those MVIs that can be created or broken by this event. In particular, for each property \(p\) (\(q\)) initiated (terminated) by events of type \(e\), CEC never takes into consideration MVIs for \(p\) (\(q\)) whose starting point (ending point) follows (precedes) \(t\). Moreover, in the case of \(p\) (\(q\)) it considers at most one MVI among those whose initiating (terminating) event precedes (follows) \(t\): It is the MVI whose starting (ending) point is closest to \(t\).

Finally, CEC preserves the basic requirements of EC of being \textit{input constructible} (each possible database state can be reached from an empty database through a suitable set of input events) and of satisfying the property of \textit{permutation of inputs} (the effect of a given set of input events does not depend on the ordering according to which they are entered in the database). As an example, CEC accepts an event occurrence that happened before some of the already acquired occurrences and revises all and only the affected MVIs. Obviously, acquiring a complete sequence of event occurrences 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 (Chittaro \textit{et al.} 1995a).

The general architecture of CEC is depicted in Fig. 1, where all its conceptual modules are highlighted. We briefly summarize here the purpose of the modules; their functioning is described in detail in the following sections. Each new event is entered into the database by \textit{update}; \textit{updateInit} and \textit{updateTermin} are then called to manage properties that are initiated and terminated by the event, respectively. In both cases, the event can either clip an existing MVI (this case is handled by \textit{breakingl} and \textit{breakingT}) or possibly create a new MVI (this case is handled by \textit{creatingl} and \textit{creatingT}). When a MVI (or a part of it) is retracted (asserted), \textit{propagateRetract} (\textit{propagateAssert}) takes care of propagating that change to properties that depend on the changed one. Propagation of assertions and retractions can recursively activate the process of breaking or creating MVIs.
CEC has been developed in a top-down fashion and extensively tested to ensure that it yields the same outputs as EC.

In the next two sections we first describe the process of breaking and creating MVIs and then the process of propagating rejections and assertions. The effects of updates are graphically depicted according to the conventions in Fig. 2.


CEC allows events to be entered in the database by means of update. It explicitly records the occurrence of the event \((E, T)\) in the database and then it calls updateInit and updateTermin.

If \(E\) initiates a property \(Prop\) at time \(T\), updateInit tests if there exists a MVI \([T_1, T_2]\) for \(Prop\) such that \(T_1 \leq T < T_2\). If such an interval exists, then \(E\) may possibly break it. On the contrary, if it does not exist, then \(T\) can be the starting point of a new MVI for \(Prop\). More specifically, these are the actions taken in each of the two cases:

1. If the interval exists, we face two possibilities, which are handled by breaking1: (i) if \(T_1 = T\), there is already an event occurring at \(T\) that initiates \(Prop\) and then no changes are needed to interval \([T_1, T_2]\) at the moment; (ii) otherwise, interval \([T_1, T_2]\) is shortened (Fig. 3) in such a way that the new starting point becomes \(T\), \(Prop\) does
not hold anymore in the clipped part \([T_1, T]\), and the retraction of \([T_1, T]\) has to be propagated.

2. If the interval does not exist, creating1 checks if a new MVI starting at \(T\) holds, looking for one of these two cases: (i) there is a “pending” terminating event for property \(Prop\) occurring at time \(T_3\) after \(T\) with no interruptions in between and thus the new MVI becomes \([T, T_3]\), (ii) there are no initiating or terminating events for property \(Prop\) occurring after \(T\) and thus the new MVI becomes \([T, \text{infPlus}]\). If the new MVI is found, it is added to the database (Fig. 4) and the new assertion has to be propagated.

Appendix B contains the code that implements the previous reasoning activities: \textit{update} is implemented by the update predicate, \textit{updateInit} by the updateInit and the insideLeft-ClosedInt predicates, \textit{breaking1} by the breaking1 predicate, \textit{creating1} by the creating1, and the newTermination predicates.

The descriptions of \textit{updateTermin}, \textit{breakingT}, and \textit{creatingT} are symmetrical to those of \textit{updateInit}, \textit{breaking1}, and \textit{creating1} and are omitted here for brevity. The effects of \textit{breakingT} and \textit{creatingT} are shown in Figs. 5 and 6.

3.2. Propagation of Retractions and Assertions

Each time a MVI \([T_1, T_2]\) 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 \([T_1, T_2]\) 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 \textit{initiates.at} or \textit{terminates.at} clauses having the retracted (asserted) property as a condition have to be reconsidered over the interval.

In the case of \textit{propagateRetract}, two different situations are distinguished: possibly invalidated initiations and possibly invalidated terminations. In the case of possibly invalidated initiations (terminations), the retracted property is a condition for the initiation (termination)

FIGURE 5. BreakingT in the case of either finite or infinite termination.
of a dependent property \( P_D \) at the occurrence of an event \( E_{P_D} \), and the occurrence time \( T_3 \) (\( T_4 \)) of \( E_{P_D} \) belongs to the retracted interval \((T_1, T_2)\). In this case, the right (left) part of \([T_1, T_2]\) overlaps the possibly affected MVI \([T_3, T_4]\) for \( P_D \).

In the case of possibly invalidated initiations, four different situations (summarized in the first column of Fig. 7) can occur:

1. independency: \( P_D \) still initiates at \( T_3 \), because there exists a successful \textit{initiates at} clause, which does not include the retracted property as condition. The MVI for \( P_D \) is thus unchanged.

2. revised initiation with finite termination: property \( P_D \) terminates at a time point \( T_4 \) (it does not hold forever) and either there exists an initiating event for \( P_D \) at a time preceding \( T_4 \) with no interruptions in between or no initiating or terminating events occur before \( T_4 \) (and \( P_D \) is assumed to hold from infMin). Resulting changes to property \( P_D \) are propagated.

3. revised initiation with infinite termination: property \( P_D \) holds until infPlus and there exists an event occurring at time \( T_5 \) that initiates \( P_D \) with \( P_D \) holding uninterrupted after \( T_5 \). In such a case, \( T_5 \) becomes the new starting point and this modification is propagated as before. Unlike situation 2, where the new starting point can also be infMin, in this situation an initiating event for \( P_D \) must be found.

4. vanishing: if none of the above three cases applies, the MVI for \( P_D \) is fully retracted and this retraction is propagated.

In the case of invalidated terminations, there are four possible situations that are symmetrical to the previously described ones and are summarized by the second column of Fig. 7.

Three main Prolog predicates implement \textit{propagateRetract}: the \textit{propagateRetract} predicate finds dependent properties \( P_D \) relevant to the interval of retraction and then uses the \textit{reviseRightOverlap} and the \textit{reviseLeftOverlap} predicates to handle possibly invalidated initiations and possibly invalidated terminations cases, respectively.

In the case of \textit{propagateAssert}, two different situations are distinguished: possibly new initiations and possibly new terminations. In the case of new initiations (terminations), the asserted property is a condition for the initiation (termination) of property \( P_D \) at the occurrence of event \( E_{P_D} \), the occurrence time \( T_3 \) (\( T_4 \)) of \( E_{P_D} \) belongs to \((T_1, T_2)\), and there is not already a MVI for \( P_D \) with \( T_3 \) (\( T_4 \)) as its starting (ending) point. In this case, \textit{updateInit} (\textit{updateTermin}) is recursively called in order to check if \( P_D \) is now initiated (terminated) at \( T_3 \) (\( T_4 \)) and possibly revising the database accordingly (Figs. 8 and 9). The \textit{propagateAssert} predicate in Appendix B implements this activity.

3.3. Supporting the Weak Interpretation in CEC

It is worth noting that supporting a weak interpretation of \textit{initiates at} in CEC is relatively straightforward. Update processing is indeed simplified, because the \textit{breaking1} and \textit{creating1} modules in Fig. 1 are no longer needed. On the other hand, in addition to the two cases of Fig. 4, \textit{creating1} has to deal with the possibility of anticipating the initiation of a MVI \([T_1, T_2]\).
when a new initiating event occurring at \( T \) (before \( T_1 \)) is recorded, and there is no known terminating event between \( T \) and \( T_1 \). Similarly, breaking does not have to necessarily retract all the clipped part \([T, T_2] \) of a broken interval, but it has to verify first if there exists an initiating event occurring at \( T_3 \) belonging to \([T, T_2] \) so that the property can be maintained over \([T_3, T_2] \). Finally, according to the weak interpretation, \textit{propagateRetract} and \textit{propagateAssert} do not have to deal with persistence in the past. The weak version of CEC has been tested on the patient monitoring domain (Chittaro \textit{et al.} 1995a). Moreover, the weak and strong interpretations can be integrated in a uniform framework as shown in (Chittaro \textit{et al.} 1995c).
3.4. Running the Lighting System Example in CEC

We now show how CEC builds and maintains the set of cached MVIs, by examining one execution of the lighting system example. Suppose we perform the following updates in the nonchronological order shown:

\[- \text{update}(\text{initially}(\text{pwrAvail}),0), \text{update}(\text{turnOff},8), \text{update}(\text{turnOn},4), \text{update}(\text{turnOn},10), \text{update}(\text{pwrFail},6), \text{update}(\text{pwrRstr},12).\]

Figure 10 illustrates the effects of each of the six updates, showing the evolution of the full set of cached MVIs. The effect of the first update is to initiate a \text{pwrAvail} property that holds between 0 and \text{infPlus} (\text{creatingI} 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 \text{lightsOn} interval that holds between \text{infMin} and 8 and (ii) a \text{switchOn} interval that also holds between \text{infMin} and 8 (\text{creatingI} 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 \text{lightsOn} is broken at 4, and validity between \text{infMin} and 4 is thus retracted and (ii) the MVI of property \text{switchOn} is also broken at 4, and validity between \text{infMin} and 4 is thus retracted (\text{breakingI} in both cases). Propagation of retraction for both properties has no effect. The effects of the fourth update are to initiate: (i) a \text{lightsOn} interval that holds between 10 and \text{infPlus} and (ii) a \text{switchOn} interval that also holds between 10 and \text{infPlus} (\text{creatingI} in both cases). Propagation of assertion for both properties has no effect. The effects of the fifth update are: (i) the MVI of property \text{pwrAvail} is broken at 6 and validity between 6 and \text{infPlus} is thus retracted (\text{breakingI}), (ii) retraction is propagated and leads to reconsider the effects of event \text{turnOn} at 10, leading to the retraction of property \text{LightsOn} holding between

![Figure 10. Trace of the lighting system example.](image-url)
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 has no effect. 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 (creating1 in both cases). Propagation of assertion for
both properties has no effect. 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 and contrast the worst-case complexities of query and update
processing in EC and CEC with a standard Prolog interpreter. Formal proofs of stated results
are given in Appendix A.

In both calculi, updates consist in the addition of a time-stamped event occurrence. With
regard to query processing, we focus on the execution of mholds_for queries returning the
full set of MVIs for a given property \( p \), and measure their complexity in terms of accesses to
database facts (happens_at in EC, and both happens_at and mholds_for in CEC). In Prolog
such queries take the following form:

\[ \text{?- bagof(MVI, mholds_for(p,MVI), MVIs).} \]

We assume that the database contains \( n \) initiating events and \( n \) terminating events for any
property. We first determine the complexity of query processing in EC devoid of context
dependency and then show how its performance heavily decreases when context dependency
is added. Then, we prove that the addition of the caching mechanism makes the cost of
query processing in CEC (with and without context dependency) linear. As usual, such an
improvement in efficiency of query processing is obtained at the expense of an increase in
complexity of update processing, but we show that the complexity of update processing in
CEC is much less than the complexity of EC query processing.

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.

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 and prove the following theorem.

Theorem 1. The complexity of query processing for EC without context dependency, mea-
sured in terms of accesses to happens_at facts, is \( O(n^3) \), where \( n \) is the number of initiating
(terminating) events in the database for the considered property.

The proof in Appendix A is performed in two steps. We first determine a cubic upper bound
\( (2n^3 + 5n^2 + 3n) \) for the total number of accesses to happens_at facts; then we show that this
cubic limit is actually reached.

To determine the complexity of query processing in EC with context dependency, we
must take into account that a condition of a context-dependent initiates_at or terminates_at
may itself be context-dependent. As a general rule, we must consider an arbitrary nesting
level of context-dependent properties. Let \( C_{bk} \) be the maximum number of conditions in an
initiates\_at or terminates\_at clause, and $L_{bk}$ 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 such that:

(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 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. The acyclicity of the graph is needed to make the comparison between EC and CEC complexities possible. 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. For example, a sufficient condition is that at each time point at most one of the properties involved in the property cycle may hold. Examples of noncritical property cycles in a medical application domain are given in Chittaro et al. (1995a).

In Appendix A, we prove the following theorem by induction on the nesting level of context-dependent properties $L_{bk}$:

**Theorem 2.** The complexity of EC query processing $\text{Comp}(\text{query}(L_{bk}))$, measured in terms of accesses to happens\_at facts, is $O(n^{(L_{bk}+1)2})$, where $n$ is the number of initiating (terminating) events in the database for the considered property.

It is worth noting that $C_{bk}$ and $L_{bk}$ can be redefined for each single property to evaluate the complexity of specific queries or classes of queries.

### 4.2. The Complexity of Query and 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. Unlike EC, where each evaluation of a mholds\_for query results in 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 the value of $L_{bk}$.

To determine the complexity of update processing in CEC, some preliminary notions are needed. First of all, let $P$ be the maximum number of properties initiated or terminated by a single event and $C_{fw}$ be the maximum number of initiates\_at and terminates\_at clauses containing the same condition. $C_{fw}$ is in general different from $C_{bk}$. As an example, consider the following set of clauses: \(\text{initiates\_at}(e1, \{p4, p5\}, p1, T) \leftarrow \ldots, \text{initiates\_at}(e2, \{p4, p6\}, p2, T) \leftarrow \ldots, \text{initiates\_at}(e3, \{p4, p5\}, p3, T) \leftarrow \ldots\). The values of $C_{bk}$ and $C_{fw}$ associated with this set are 2 (each property has two conditions) and 3 (property $p4$ is a condition for $p1$, $p2$, and $p3$), respectively. We must also 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. In general, we must consider an arbitrary level of propagation of assertions (retractions). Let $L_{tw}$ be the maximum level of propagation from a single property.
The Orders of Complexity of Update and Query Processing in EC and CEC.

<table>
<thead>
<tr>
<th>$L_{bk}$ = $L_{fw}$</th>
<th>EC update</th>
<th>EC query</th>
<th>CEC update</th>
<th>CEC query</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>const</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>1</td>
<td>const</td>
<td>$O(n^6)$</td>
<td>$O(n^4)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>2</td>
<td>const</td>
<td>$O(n^9)$</td>
<td>$O(n^4)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$k$ ($&gt;0$)</td>
<td>const</td>
<td>$O(n^{(k+1):3})$</td>
<td>$O(n^{(k+1):2})$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

$L_{fw}$ can be formally defined as the length of the longest path in the property dependency graph, and then it is equal to $L_{bk}$ (obviously, if $L_{bk}$ and $L_{fw}$ are redefined to account for nesting and propagation levels of specific properties, they are not necessarily equal anymore).

In Appendix A, we prove the following lemma by induction on the propagation level $L_{fw}$:

**Lemma 1.** For each $L_{fw} > 0$, the complexity of propagating assertions, measured in terms of accesses to `happens_at` and `mholds_for` facts, is $O(n^{L_{fw}+3})$, where $n$ is the number of initiating (terminating) events in the database for the considered property. This complexity is equal to the complexity of propagating rejections.

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 not greater than the cost of `propagateAssert`. Finally, we provide the general cost expressions of `propagateRetract` and `propagateAssert`.

On the basis of this lemma, we prove the following theorem about the complexity of update processing in CEC.

**Theorem 3.** The complexity of CEC update processing $\text{Comp}(update(L_{fw}))$, measured in terms of accesses to `happens_at` and `mholds_for` facts, is $O(n^2)$ when $L_{fw} = 0$ and $O(n^{(L_{fw}+1)^2})$ otherwise, where $n$ is the number of initiating (terminating) events in the database for the considered property.

The orders of complexity of query and update processing in EC and CEC are summarized in Table 1.

5. CONCLUSIONS AND CURRENT WORK

In this article, we dealt with the problem of providing EC with an efficient implementation. Since we aimed at being beneficial to both the theoretically minded and the implementation-oriented communities, we provided both a proposal for a cached version of EC and a mathematical analysis of the computational complexity of EC and its cached version. We extensively tested CEC on toy examples (such as the lighting system) and we experimented it on a real-world patient monitoring problem (Chittaro et al. 1995).

Currently, we are investigating several topics related to those discussed in this article. In particular, we are exploring the problem of reasoning about partially ordered sequences of events (Cervesato et al. 1993, 1995; Chittaro et al. 1994, 1995b). In this setting, our aims are: (i) to propose variants of EC that support queries about which MVI must certainly hold
and which are possible but not certain, (ii) to give them a uniform modal logic interpretation, (iii) to devise an efficient implementation for them following the same line proposed in this article. Moreover, while this article is devoted to the more complex strong interpretation of EC, we are also experimenting with the simpler weak interpretation. Experimental results on the patient monitoring problem show that in the average case, “weak” CEC is more efficient than “strong” CEC as soon as the number of events is sufficiently high, regardless of the order of their acquisition. Finally, we are evaluating the possibility of further increasing efficiency and guaranteeing response within a given time when events are acquired according to their chronological ordering, by deleting the oldest, no more necessary events and MVIs from the active CEC database (forgetting mechanism). Deleted items can be moved for historical purposes into a separate database that can be also queried with CEC.

ACKNOWLEDGMENTS

This work has been partially supported by the P.A.O.L.A. Consortium (ASEM, INSIEL, and Università di Udine). The authors would like to thank one of the anonymous reviewers of TIME '94 for the helpful comments. This work benefited also from the discussions with some of the TIME '94 participants (in particular, H. Hamilton, M. Tambe, and A. Trudel) on the workshop version of this paper.

REFERENCES

APPENDIX A

Proof of Theorem 1. By hypothesis, there are $n$ initiating events and $n$ terminating events for the given property in the database. Since Prolog adopts the first-left selection rule, during the exploration of the search tree for the query the test on event ordering in clause (1.1) of $\text{not holds for}$ is performed only after the execution of the $\text{initiates at}$ and $\text{terminates at}$ predicates. Moreover, since the $\text{initiates at}$ and $\text{terminates at}$ predicates in the body of clause (1.1) are invoked with the temporal argument unbound, for each access to one of the $n$ initiating events there are $n$ accesses to terminating events. Therefore, the total number of accesses to $\text{happens at}$ facts is $n(1 + n)$, where $n$ is the number of initiating events in the database and $1 + n$ is the number of accesses to $\text{happens at}$ facts for each initiating event, i.e., an access to the initiating event itself followed by $n$ accesses to terminating events.

The temporal ordering of events does not influence the total number of accesses to $\text{happens at}$ facts, but it determines the number of pairs of initiating and terminating events that pass the $\text{End gt Start}$ test. In the worst case, every pair of initiating and terminating events passes the $\text{End gt Start}$ test and is then picked up for the subsequent processing. This is the case, for instance, of an (incomplete) narrative consisting of a sequence of $n$ chronologically ordered initiating events followed by a sequence of $n$ chronologically ordered terminating events. The number of ($\text{initiating-event, terminating-event}$) pairs is $n \cdot n$ and all of them pass the $\text{End gt Start}$ test. However, also in the case of a (complete) narrative consisting of a sequence of $n$ chronologically ordered initiating events each one immediately followed by its corresponding terminating event, the number of pairs that pass the $\text{End gt Start}$ test stays quadratic (it is equal to $n(n + 1)/2$).

For each pair passing the test, clause (1.1) then executes a $\text{not broken during}$ query with both arguments instantiated. The complexity of this query is linear, the worst case $(2n)$ given by a pair consisting of an initiating event immediately followed by a terminating event. In such a case, all $n$ occurrences of terminating events and all $n$ occurrences of initiating events for the considered property are checked against the pair. An upper bound for the total number of accesses to $\text{happens at}$ facts during the execution of the $\text{not broken during}$ in clause (1.1) is then $n^22n$, where $n^2$ is the maximum number of pairs possibly passing the $\text{End gt Start}$ test and $2n$ is the maximum number of accesses to $\text{happens at}$ facts possibly occurring for each pair. Merging the previous results, a cubic upper bound equal to $n(1 + n) + n^22n = 2n^3 + n^2 + n$ can be established for the total number of accesses to $\text{happens at}$ facts during
the execution of clause (1.1).³ Continuing the exploration of the search tree, Prolog then tries clause (1.2) and (1.3) of mholds.four. These two clauses make \( n \) accesses to initiating and terminating events, respectively. For each accessed event, both clauses execute a not broken.during query with both arguments instantiated. A quadratic upper bound equal to 
\[
(n(1 + 2n) + n(1 + 2n) = 2n(1 + 2n) = 4n^2 + 2n)
\]
can be thus established for the total number of accesses to happens.at facts during the execution of both clause (1.2) and (1.3). Therefore, an upper bound for the total number of accesses to happens.at facts during the execution of all the three clauses is 
\[
2n^3 + 5n^2 + 3n.
\]
We show in the following that the cubic limit is actually reached.

Consider the above-mentioned case of a sequence of \( n \) chronologically ordered initiating events followed by a sequence of \( n \) chronologically ordered terminating events, recorded in the Prolog database according to their chronological order. We have already shown that the number of (initiating-event, terminating-event) pairs passing the End gt Start test in clause (1.1) is \( n \cdot n \), and the corresponding number of accesses is \( n(1 + n) \). We determine now the complexity of executing the not broken.during occurring in the body of clause (1.1).

Let us denote initiating and terminating events by \( e_{i1}, \ldots, e_{in} \) and \( e_{t1}, \ldots, e_{tn} \), respectively. Considering clause (1.1), for each pair \( (e_{i1}, e_{tj}) \), with \( j = 1, \ldots, n \), the not broken.during query involves two accesses to happens.at facts, i.e., the accesses to \( e_{i1} \) and \( e_{t1} \); for each pair \( (e_{i2}, e_{tj}) \), with \( j = 1, \ldots, n \), the not broken.during query involves three accesses to happens.at facts, i.e., the accesses to \( e_{i1}, e_{t2} \) and to \( e_{t1} \); and so on. As a general rule, for each pair \( (e_{ik}, e_{tj}) \), with \( k = 1, \ldots, n - 1 \) and \( j = 1, \ldots, n \), the not broken.during query involves \( k + 1 \) accesses to happens.at facts. The same rule applies for each pair \( (e_{in}, e_{tj}) \), with \( j = 2, \ldots, n \), given that the not broken.during query involves \( n + 1 \) accesses to happens.at facts. For the pair \( (e_{in}, e_{t1}) \), the not broken.during query succeeds and, then, it involves \( 2n \) accesses to happens.at facts. The resulting number of accesses to happens.at facts is

\[
2n + 3n + \cdots + nn + (n + 1)(n - 1) + 2n = (n^3 + 3n^2 + 2n - 2)/2.
\]

The total number of accesses to happens.at facts for clause (1.1) is thus equal to

\[
(n^3 + 3n^2 + 2n - 2)/2 + n^2 + n = (n^3 + 5n^2 + 4n - 2)/2.
\]

It is possible to show that the complexity of executing the not broken.during is cubic also in the case of the complete narrative consisting of \( n \) initiating events each one followed by its corresponding terminating event (it is equal to \((n^3 + 9n^2 + 2n)/3)\).⁴

Proof of Theorem 2. If we consider \( L_{bk} = 0 \), we fall in the already discussed case of the basic calculus, whose complexity has been shown to be \( O(n^3) \). When \( L_{bk} \) is 1 or more, the evaluation of each initiates.at or terminates.at predicates results in the evaluation of at worst \( C_{bk} \) conditions and then the evaluation of each of these conditions results in the evaluation of a mholds.for predicate with the temporal argument unbound and a \( L_{bk} - 1 \) nesting level. The relationship between the complexity upper bounds for query(\( L_{bk} \)) and query(\( L_{bk} - 1 \)) is expressed by the recurrent expression

\[
\text{Comp(query}(L_{bk})) = (2n^3 + 5n^2 + 3n)(1 + C_{bk} \cdot \text{Comp(query}(L_{bk} - 1))))
\]

where \( 2n^3 + 5n^2 + 3n \) is the number of initiates.at or terminates.at involved in the evaluation of mholds.for (as in the context-independent case), each of them recursively leading to the

³In the case of the weak interpretation of initiates.at and terminates.at, the upper bound stays cubic; the only difference is that \( 2n^3 \) is replaced by \( n^3 \) given that initiating events are not interrupting.

⁴It goes without saying that this complexity can be lower for some other narratives. As an example, it is quadratic in the case of a narrative where all terminating events precede all initiating events, because no event pair passes the End gt Start test.
evaluation of at most $C_{bk}$ conditions. On the basis of such a relationship, we prove by induction on the nesting level $L_{bk}$ that

$$\text{Comp}(\text{query}(L_{bk})) = \left(\sum_{i=1}^{L_{bk}} C_{bk}^{i-1} (2n^3 + 5n^2 + 3n)^i\right) + C_{bk}^{L_{bk}} (2n^3 + 5n^2 + 3n)^{L_{bk}+1}$$

and thus, that the complexity of $\text{query}(L_{bk})$ is $O(n^{(L_{bk}+1)^3})$.

This expression is trivially true for $L_{bk} = 0$. Suppose now that it is true for $L_{bk} - 1$; its truth for $L_{bk}$ can be proved as follows:

$$\text{Comp}(\text{query}(L_{bk})) = (2n^3 + 5n^2 + 3n)(1 + C_{bk} \cdot \text{Comp}(\text{query}(L_{bk} - 1)))$$

$$= (2n^3 + 5n^2 + 3n)(1 + C_{bk}\left(\sum_{i=1}^{L_{bk} - 1} C_{bk}^{i-1} (2n^3 + 5n^2 + 3n)^i\right))$$

$$+ C_{bk}^{L_{bk}-1} (2n^3 + 5n^2 + 3n)^{L_{bk}})$$

$$= \left(\sum_{i=1}^{L_{bk}} C_{bk}^{i-1} (2n^3 + 5n^2 + 3n)^i\right) + C_{bk}^{L_{bk}} (2n^3 + 5n^2 + 3n)^{L_{bk}+1}.$$}

Proof of Lemma 1. Step 1: First, for $L_{fw} = 0$ we have

$$\text{Comp}(\text{propagateRetract}(0)) = \text{Comp}(\text{propagateAssert}(0)) = 0$$

because in such a case there is no propagation of assertions and retractions. For each $L_{fw} \geq 1$, the cost of $\text{propagateRetract}(L_{fw})$ is the cost of revising relevant MVIs that right overlap or left overlap the retracted interval, and it is given by the expression

$$\text{Comp}(\text{propagateRetract}(L_{fw})) = C_{fw}n(1 + 1 + \text{Comp}(\text{reviseRightOverlap}(L_{fw} - 1)))$$

$$+ 1 + 1 + \text{Comp}(\text{reviseLeftOverlap}(L_{fw} - 1))).$$

where $C_{fw}n$ is an upper bound to the number of initiating (terminating) events for properties (c-properties) having the considered property as condition, and the unitary factors refer to the $\text{happens.at}$ and $\text{rightOverlap}$ (leftOverlap) invocations inside $\text{propagateRetract}$.

Let $I_r$ be the retracted interval and $I_p$ be a MVI of a c-property such that $I_p$ right overlaps $I_r$. The retraction of $I_r$ leaves $I_p$ unchanged (independency), modifies its starting point (revisedInitiation), or forces its full retraction (vanishing). The worst-case cost of reviseRightOverlap($L_{fw} - 1$) is given by the most expensive of these three alternatives.

The cost of $\text{independency}$ is the cost of a successful evaluation of $\text{initiates.at}$ with both the property and the temporal argument instantiated, which is equal to

$$\text{Comp}(\text{independency}(L_{fw} - 1)) = 1 + C_{bk}n.$$
the resulting cost is the sum of the cost of determining the new starting point of the revised
interval (newInitiation):

\[
\text{Comp}(\text{newInitiation}) = n(1 + C_{bk}n + 2n(1 + C_{bk}n)) + 2n(1 + C_{bk}n)
\]

and the maximum between the costs of assertion and retraction propagations with nesting
level \( L_{fw} - 1 \). In case (ii), the resulting cost is given by the cost of \( n - 1 \) successful evaluations
of \( \text{initiates.at} \) with only the property argument instantiated, each one followed by a failing
evaluation of \( \text{not broken during} \)

\[
(n - 1)(1 + C_{bk}n + 2n(1 + C_{bk}n))
\]

plus the cost of a successful evaluation of \( \text{initiates.at} \) followed by a successful evaluation of
\( \text{not broken during} \):

\[
1 + C_{bk}n + 2n(1 + C_{bk}n)
\]

plus the maximum between the costs of assertion and retraction propagations with nesting
level \( L_{fw} - 1 \). This maximum is common to (i) and (ii), and thus it can be factorized out.
The resulting expression can then be simplified, reducing to the cost expression

\[
\text{Comp}(\text{revisedInitiation}(L_{fw} - 1)) = 2C_{bk}n^3 + (3C_{bk} + 2)n^2 + (C_{bk} + 3)n + 1 + \max(\text{Comp}(\text{propagateRetract}(L_{fw} - 1)), \text{Comp}(\text{propagateAssert}(L_{fw} - 1)))
\]

The cost of \( \text{vanishing} \) is the cost of a failing evaluation of the \( \text{revisedInitiation} \) branch:

\[
2C_{bk}n^3 + (3C_{bk} + 2)n^2 + (C_{bk} + 3)n + 1
\]

plus the cost of propagation of retractions with nesting level \( L_{fw} - 1 \).

It is easy to see that the cost of \( \text{independency} \) is always less than the costs of the other
two alternatives and that the cost of \( \text{vanishing} \) is not greater than the cost of \( \text{revisedInitiation} \).
This allows us to conclude that the cost of \( \text{reviseRightOverlap}(L_{fw} - 1) \) is equal to that of
\( \text{revisedInitiation}(L_{fw} - 1) \).

The worst-case cost analysis for \( \text{reviseLeftOverlap}(L_{fw} - 1) \) is perfectly symmetric, and
it results in the same cost expression. Thus, the cost of \( \text{propagateRetract}(L_{fw}) \) is given by
the following expression:

\[
\text{Comp}(\text{propagateRetract}(L_{fw})) = 2C_{fw}n^3 + (3C_{bk} + 2)n^2 + (C_{bk} + 3)n + 3 + \max(\text{Comp}(\text{propagateAssert}(L_{fw} - 1)), \text{Comp}(\text{propagateRetract}(L_{fw} - 1)))
\]

For each \( L_{fw} \geq 1 \), the cost of \( \text{propagateAssert}(L_{fw}) \) is given by the expression

\[
\text{Comp}(\text{propagateAssert}(L_{fw})) = C_{fw}n(1 + \text{Comp}(\text{updateInit}(L_{fw} - 1)) + 1 + \text{Comp}(\text{updateTerm}(L_{fw} - 1)))
\]

where \( C_{fw}n \) is the upper bound to the number of initiating (terminating) events for properties
having the considered property as condition (\( c \)-properties).

For each \( c \)-property \( p \), the assertion of a new interval \( I \) causes either the shortening of
an existing MVI or the creation of a new MVI, or has no effect at all. \( \text{updateInit}(L_{fw} - 1) \)
and \( \text{updateTerm}(L_{fw} - 1) \) respectively deal with initiating and terminating events for \( p \). The worst-case cost of evaluating \( \text{updateInit}(L_{fw} - 1) \) with all its arguments instantiated is given by the cost of a successful evaluation of \( \text{initiatesAt} \) with both the property and the temporal argument instantiated \((1 + C_{bk}n)\), plus the cost of evaluating \( \text{insideLeftClosedInt} \) (that is, \( n \)), plus the maximum between the cost of shortening an existing MVI and the cost of creating a new MVI. The cost of shortening an existing MVI is the cost of propagating the retraction:

\[
\text{Comp}(\text{breakingl}(L_{fw} - 1)) = \text{Comp}(\text{propagateRetract}(L_{fw} - 1)).
\]

The cost of creating a new MVI is the cost of determining its ending point:

\[
\begin{align*}
\text{Comp}(\text{newTermination}) &= \text{Comp}(\text{newInitiation}) \\
&= n(1 + C_{bk}n + 2n(1 + C_{bk}n)) + 2n(1 + C_{bk}n)
\end{align*}
\]

plus the cost of propagating its assertion. Therefore, the total cost is

\[
\begin{align*}
\text{Comp}(\text{creatingl}(L_{fw})) &= 2C_{bk}n^3 + (3C_{bk} + 2)n^2 \\
&\quad + 3n + \text{Comp}(\text{propagateAssert}(L_{fw})).
\end{align*}
\]

The resulting cost expression for \( \text{updateInit}(L_{fw} - 1) \) is

\[
\begin{align*}
\text{Comp}(\text{updateInit}(L_{fw} - 1)) &= 1 + C_{bk}n + n + \max(\text{Comp}(\text{propagateRetract}(L_{fw} - 1)), 2C_{bk}n^3 \\
&\quad + (3C_{bk} + 2)n^2 + 3n + \text{Comp}(\text{propagateAssert}(L_{fw} - 1))).
\end{align*}
\]

The worst-case cost analysis for \( \text{updateTerm}(L_{fw} - 1) \) is perfectly symmetric, and it results in the same cost expression. Therefore, the cost of \( \text{propagateAssert}(L_{fw}) \) is given by the following expression:

\[
\begin{align*}
\text{Comp}(\text{propagateAssert}(L_{fw})) &= 2C_{fw}n(2 + (C_{bk} + 1)n + \max(\text{Comp}(\text{propagateRetract}(L_{fw} - 1)), 2C_{bk}n^3 \\
&\quad + (3C_{bk} + 2)n^2 + 3n + \text{Comp}(\text{propagateAssert}(L_{fw} - 1)))).
\end{align*}
\]

Step 2: On the basis of the recurrent cost expressions for \( \text{propagateAssert} \) and \( \text{propagateRetract} \) obtained in the first step, we prove that for each \( L_{fw} \geq 0 \):

\[
\text{Comp}(\text{propagateRetract}(L_{fw})) \leq \text{Comp}(\text{propagateAssert}(L_{fw})).
\]

The proof is by induction on \( L_{fw} \). The inequality is trivially true for \( L_{fw} = 0 \). Suppose now that it is true for \( L_{fw} - 1 \); its truth for \( L_{fw} \) can be proved as follows.

First of all, according to the inductive hypothesis the cost of \( \text{propagateAssert} \) for \( L_{fw} \) is given by the expression:

\[
\begin{align*}
\text{Comp}(\text{propagateAssert}(L_{fw})) &= 2C_{fw}n(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 \\
&\quad + 2C_{bk}n^3 + \text{Comp}(\text{propagateAssert}(L_{fw} - 1))).
\end{align*}
\]

Similarly, from the inductive hypothesis it follows that the cost of \( \text{propagateRetract} \) for \( L_{fw} \) is given by the expression

\[
\begin{align*}
\text{Comp}(\text{propagateRetract}(L_{fw})) &= 2C_{fw}n(3 + (C_{bk} + 3)n + (3C_{bk} + 2)n^2 \\
&\quad + 2C_{bk}n^3 + \text{Comp}(\text{propagateAssert}(L_{fw} - 1))).
\end{align*}
\]
It is trivial to prove that: (i) for \( n = 1 \), the complexity of \( \text{propagateRetract} \), with propagation level \( L_{fw} \), is equal to the complexity of \( \text{propagateAssert} \), with the same propagation level, and (ii) for \( n > 1 \), the complexity of \( \text{propagateRetract} \) is strictly less than the complexity of \( \text{propagateAssert} \).

Step 3: By induction on the nesting level \( L_{fw} \) we can easily show that the cost expression for \( \text{propagateAssert}(L_{fw}) \) is

\[
\text{Comp}(\text{propagateAssert}(L_{fw})) = 2C_{fw}n(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3) \\
+ (2C_{fw}n)^2(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 \\
+ 2C_{bk}n^3) + \cdots + (2C_{fw}n)^{L_{fw}} \\
\times (2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3),
\]

which can be abbreviated as

\[
\text{Comp}(\text{propagateAssert}(L_{fw})) = \sum_{i=1}^{L_{fw}} (2C_{fw}n)^i(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3).
\]

In a similar way, it can be easily proved by induction that the cost expression for \( \text{propagateRetract}(L_{fw}) \) is

\[
\text{Comp}(\text{propagateRetract}(L_{fw})) = 2C_{fw}n(3 + (C_{bk} + 3)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3) \\
+ \sum_{i=1}^{L_{fw}-1} (2C_{fw}n)^i(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3)).
\]

Proof of Theorem 3. The cost of entering a new event in the database is the cost of introducing the new MVIs it establishes as well as of clipping the stored MVIs it affects. The update procedure encompasses the effects of the event on both the initiation and termination of properties and its complexity is given by the expression:

\[
\text{Comp}(\text{update}(L_{fw})) = \text{Comp}(\text{updateInit}(L_{fw})) + \text{Comp}(\text{updateTerm}(L_{fw})).
\]

Let \( P_1 \) and \( P_2 \) (with \( P = P_1 + P_2 \)) be the number of properties initiated and terminated by the event, respectively. The costs of \( \text{updateInit}(L_{fw}) \) and \( \text{updateTerm}(L_{fw}) \) are

\[
\text{Comp}(\text{updateInit}(L_{fw})) = P_1((1 + C_{bk}n) + n + \max(\text{Comp}(\text{breaking}\_\text{init}(L_{fw})), \text{Comp}(\text{creating}\_\text{init}(L_{fw}))))
\]

and

\[
\text{Comp}(\text{updateTerm}(L_{fw})) = P_2((1 + C_{bk}n) + n + \max(\text{Comp}(\text{breaking}\_\text{term}(L_{fw})), \text{Comp}(\text{creating}\_\text{term}(L_{fw})))),
\]

respectively. For each property \( p \) belonging to the \( P_1 \) (\( P_2 \)) properties initiated (terminated) by the input event, \( 1 + C_{bk}n \) is the cost of one access to \text{happens\_at} plus the cost of evaluating \( C_{bk} \) conditions, while \( n \) is the cost of evaluating \text{inside\_Left\_Close\_Init} (\text{inside\_Right\_Close\_Init}) with the first two arguments instantiated. All MVIs for \( p \) are accessed, but at most one passes the ordering tests, because the occurrence time of the input event belongs to at most one of them.
This implies that either the breaking\textsuperscript{1} (breaking\textsuperscript{T}) or the creating\textsuperscript{1} (creating\textsuperscript{T}) predicate is evaluated only once and then its complexity is added to and not multiplied by \( n \).

Unfolding the cost expressions for breaking\textsuperscript{1} (creating\textsuperscript{1}) and creating\textsuperscript{1} (creating\textsuperscript{T}) allows us to express the cost of \textit{update(L}_{fw})\textit{ in terms of the cost of propagation and retraction of assertions:}

\[
\text{Comp}(\text{update}(L_{fw})) = P(1 + (C_{bk} + 1)n + \max(\text{Comp}(\text{propagateRetract}(L_{fw}))), 2C_{bk}n^3 + (3C_{bk} + 2)n^2 + 3n + \text{Comp}(\text{propagateAssert}(L_{fw})))) .
\]

If \( L_{fw} = 0 \) then 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 \textit{mholds} for facts. Since \( C_{bk} = 0 \), it follows that the order of complexity is \( O(n^2) \). If \( L_{fw} > 0 \), on the basis of the previous lemma we replace the max expression with its value obtaining the following expression:

\[
\text{Comp}(\text{update}(L_{fw})) = P\left(1 + (C_{bk} + 1)n + 2C_{bk}n^3 + (3C_{bk} + 2)n^2 + 3n + \sum_{i=1}^{L_{fw}}((2C_{fw}n)^i(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3)))\right),
\]

which is equivalent to

\[
\text{Comp}(\text{update}(L_{fw})) = P\left(\sum_{i=0}^{L_{fw}}((2C_{fw}n)^i(2 + (C_{bk} + 4)n + (3C_{bk} + 2)n^2 + 2C_{bk}n^3)) - 1\right).
\]

This allows us to conclude that the complexity of update processing is \( O(n(L_{fw}+1)+2) \).

\textbf{APPENDIX B}

Prolog predicates are listed in alphabetical order in the following.

\begin{verbatim}
:-
breakingI(_,T1,[T1,_]):-!.

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

:-
breakingT(_,T2,[_,T2]):-!.

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

:-
creatingI(Prop,T):-
    newTermination(Prop,[T,NewEnd]),
    assert(mholds_for(Prop,[T,NewEnd])),
    propagateAssert([T,NewEnd],Prop).
\end{verbatim}
EFFICIENT TEMPORAL REASONING IN THE CACHED EVENT CALCULUS

newInitiation(Prop, [T1, T2]) :-
    memberchk(Prop, NewStart), NewStart !< T1 !< T2.

%---------------------------------------------------------------------------------
% possibly new initiations
%---------------------------------------------------------------------------------
propagateAssert([T1, T2], AssertedProp) :-
    happens_at(EPd, T), T !< T2, !+mholds_for(Prop, [T, -]),
    updateInit(EPd, T, Pd), fail.

%---------------------------------------------------------------------------------
% possibly new terminations
%---------------------------------------------------------------------------------
propagateAssert(Prop, [T, infPlus]) :-
    \+broken_during(Prop, [T, infMin]),
    memberchk(Prop, NewStart).

newTermination(Prop, [T, NewEnd]) :-
    terminates_at(_, Prop, NewEnd), T !< NewEnd.
    \+broken_during(Prop, [T, NewEnd]), !.

newTermination(Prop, [T, infPlus]) :-
    \+broken_during(Prop, [T, infPlus]),
    memberchk(Prop, NewStart).

%---------------------------------------------------------------------------------
propagateRetract([T1, T2], RetractedProp) :- %possibly new terminations
    happens_at(EPd, T), T !< T2, !+mholds_for(Prop, [T, -]),
    updateTermin(EPd, T, Pd), fail.

%---------------------------------------------------------------------------------
propagateRetract([T1, T2], RetractedProp) :- %possibly invalidated initiations
    happens_at(EPd, T), rightOverlap([T1, T2], Prop, [T3, T4]),
    reviseRightOverlap(EPd, [T, T4]), fail.

%---------------------------------------------------------------------------------
propagateRetract([T1, T2], RetractedProp) :- %possibly invalidated terminations
    happens_at(EPd, T), leftOverlap([T1, T2], Prop, [T3, T4]),
    reviseLeftOverlap(EPd, [T3, T4]), fail.

%---------------------------------------------------------------------------------
reviseLeftOverlap(Pd,[_,T4]):- %independency
    terminates_at( _, _, Pd, T4), !.
    reviseLeftOverlap(Pd,[T3,T4]):- %revised termination with finite initiation
    T3 \= infMin, newTermination(Pd,[T3,NewEnd]), !.
    retract(mholds_for(Pd,[T3,T4])), assert(mholds_for(Pd,[T3,NewEnd])),
    ((T4 gt NewEnd) -> propagateRetract([NewEnd,T4],Pd);
     propagateAssert([T4,NewEnd],Pd)).

    reviseLeftOverlap(Pd,[infMin,T4]):- %revised termination with infinite initiation
    terminates_at( _, _, Pd, T5), \|-broken_before(Pd,T5), !.
    retract(mholds_for(Pd,[infMin,T4])), assert(mholds_for(Pd,[infMin,T5])),
    ((T4 gt T5) -> propagateRetract([T5,T4],Pd);
     propagateAssert([T4,T5],Pd)).

    reviseLeftOverlap(Pd,[T3,T4]):- %vanishing
        retract(mholds_for(Pd,[T3,T4])), propagateRetract([T3,T4],Pd).

reviseRightOverlap(Pd,[T3,infPlus]):- %revised initiation with infinite termination
    initiates-at( _, _, _, Pd, T3), !.
    initiates-at(T3,_,_,Pd), \|-broken_during(Pd,[T5,infPlus]), !.
    retract(mholds_for(Pd,[T3,infPlus])), assert(mholds_for(Pd,[T5,infPlus])),
    ( (T3 lt T5) -> propagateRetract([T5,T3],Pd);
     propagateAssert([T5,T3],Pd)).

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

updateTermin(E,T,Prop):-
    terminates_at(E,_,Prop,T),
    (insideRightCloseInt(Prop,T,[T1,T2]) -> breakingT(Prop,T,[T1,T2]);
     creatingT(Prop,T)).