Run-Time Composite Event Recognition

Alexander Artikis[1], Marek Sergot[2] and Georgios Paliouras[1]
[1]Institute of Informatics & Telecommunications,
National Centre for Scientific Research (NCSR) "Demokritos", Athens 15310, Greece
[2]Department of Computing, Imperial College London, UK
{a.artikis, paliourg}@iit.demokritos.gr, m.sergot@imperial.ac.uk

ABSTRACT

Events are particularly important pieces of knowledge, as they represent activities of special significance within an organisation: the automated recognition of events is of utmost importance. We present RTEC, an Event Calculus dialect for run-time event recognition and its Prolog implementation. RTEC includes a number of novel techniques allowing for efficient run-time recognition, scalable to large data streams. It can be used in applications where data might arrive with a delay from, or might be revised by, the underlying event sources. We evaluate RTEC using a real-world application.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms

Languages

Keywords

pattern matching, event processing, event calculus

1. INTRODUCTION

Systems for symbolic event recognition ('event pattern matching') accept as input a stream of time-stamped simple, derived events (SDE). A SDE (or 'low-level event', 'short-term activity') is the result of applying a computational derivation process to some other event, such as an event coming from a sensor [19]. Using SDE as input, event recognition systems identify composite events (CE) of interest—collections of events that satisfy some pattern. The 'definition' of a CE (or 'high-level event', 'long-term activity', 'situation' [1]) imposes temporal and, possibly, atemporal constraints on its subevents, that is, SDE or other CE.

Numerous recognition systems have been proposed in the literature—see [8, 22] for two recent surveys. Recognition systems with a logic-based representation of CE definitions, in particular, have recently been attracting attention. They exhibit a formal, declarative semantics, in contrast to other types of recognition system that often rely on an informal and/or procedural semantics. Cugola and Margara [7], for example, point out that almost all 'complex event processing languages' and several 'data stream processing languages' lack a rigorous, formal semantics. Eckert and Bry [12] note that the semantics of 'event query languages' are often somewhat ad hoc, unintuitive and generally have an algebraic and less declarative flavour. Paschke and Kozlenkov [22] state that the classical and most commercial 'production rule languages' lack a declarative semantics.

Note that logic-based CE recognition systems may be used in combination with existing non-logic-based event processing infrastructures and middleware (see [21] for an example).

Non-logic-based CE recognition systems have proven to be, overall, more efficient than logic-based ones and, thus, most industrial applications employ the former type of system. To address this issue, we present an efficient dialect of the Event Calculus (EC) [15], called 'Event Calculus for Run-Time reasoning' (RTEC). EC is a logic programming language for representing and reasoning about events and their effects. In addition to inheriting the aforementioned benefits of logic-based approaches, EC is a good candidate for CE recognition for the following reasons. First, it has built-in axioms for complex temporal representation, including the formalisation of inertia, which allow for succinct CE definitions and thus code maintenance. Second, it has direct routes to machine learning. Inductive logic programming techniques, such as [23], may be used to facilitate the construction of CE definitions. Third, EC has direct routes to reasoning under uncertainty. Probabilistic frameworks, such as [14], may be employed to address issues like noisy SDE streams and imprecise knowledge of CE definitions.

RTEC includes a number of novel implementation techniques designed to support efficient CE recognition, scalable to large SDE and CE volumes. A form of caching stores the results of sub-computations in computer memory to avoid unnecessary recomputations. A set of interval manipulation constructs simplify CE definitions and improve reasoning efficiency. A simple indexing mechanism means that RTEC is only slightly affected by SDE that are irrelevant to the CE we want to recognise and so can operate without additional SDE filtering modules. Finally, a 'windowing' mechanism supports real-time CE recognition. RTEC remains efficient and scalable in applications where SDE arrive with a (variable) delay from, or are revised by, the underlying SDE de-

DEBS '12, July 16–20, 2012, Berlin, Germany.

tection system: RTEC can update the already recognised CE, and recognise new CE, when SDE arrive with a delay or following correction or revision.

We evaluate RTEC experimentally using a real-world application: event recognition for city transport management (CTM). The code of RTEC, the CTM CE definition library, and the datasets on which the experimental evaluation was performed, are directly available from the authors.

The paper is organised as follows. Section 2 describes CTM. Section 3 outlines the RTEC representation with some examples from CTM. Section 4 presents the algorithms. An experimental evaluation is given in Section 5. In Section 6 we compare RTEC with related work, and in Section 7 we summarise and outline further directions.

2. CITY TRANSPORT MANAGEMENT

In the context of PRONTO project we are developing a recognition system to support city transport management (CTM).[1] The system is being tested in the city of Helsinki, Finland. Buses and trams are equipped with in-vehicle sensors that send measurements such as GPS coordinates, acceleration information, in-vehicle temperature and noise levels to a central server, providing information about the current status of the transport system (for example, the location of buses and trams on the city map). Given the SDE extracted from these sensors, and from other data sources such as digital maps, CE are recognised related to the punctuality of a vehicle, passenger and driver comfort, passenger and driver safety, and passenger satisfaction, among others. The recognised CE are made available to the transport control centre in order to facilitate resource management. The choice of CE, and their definitions in terms of SDE, were specified by the domain experts (end users).

3. EVENT CALCULUS

Our CE recognition system is a logic programming (Prolog) implementation of an Event Calculus (EC) dialect. EC [15] is based on a many-sorted, first-order predicate calculus, and is used for representing and reasoning about events and their effects. For the dialect introduced here, RTEC, the time model is linear and includes integers. Where F is a *fluent*—a property that is allowed to have different values at different points in time—the term $F = V$ denotes that fluent F has value V. Boolean fluents are a special case in which the possible values are true and false. Informally, $F = V$ holds at a particular time-point if $F = V$ has been *initiated* by an event at some earlier time-point, and not *terminated* by another event in the meantime (law of inertia).

Following Prolog's syntax, variables start with an uppercase letter (and are universally quantified, unless otherwise indicated) while predicates and constants start with a lowercase letter. The holdsAt predicate is used to express that a fluent has a particular value at a given time. An instance of an event type is denoted by means of happensAt. For example, happensAt($temperature\_change(11, bus, cold)$, 5) represents the occurrence of event type $temperature\_change(11, bus, cold)$ at time-point 5. When it is clear from context, we do not distinguish between an event (fluent) and its type. As in other versions of EC, an *event description* in RTEC includes axioms that define the event instances (with the use of the happensAt predicate), the effects of events (with the use of the initiatedAt and terminatedAt predicates), and the values of the fluents (with the use of the initially, holdsAt and holdsFor predicates), as well as other, possibly atemporal, information. Table 1 summarises the RTEC predicates. The last three items in the table are interval manipulation predicates specific to RTEC.

[1] http://www.ict-pronto.org/

Table 1: Main predicates of RTEC.

Predicate	Meaning
happensAt(E, T)	Event E is occurring at time T
initially($F = V$)	The value of fluent F is V at time 0
holdsAt($F = V$, T)	The value of fluent F is V at time T
holdsFor($F = V$, I)	I is the list of maximal intervals for which $F = V$ holds continuously
initiatedAt($F = V$, T)	At time T a period of time for which $F = V$ is initiated
terminatedAt($F = V, T$)	At time T a period of time for which $F = V$ is terminated
union_all(L, I)	I is the list of maximal intervals produced by the union of the lists of maximal intervals of list L
intersect_all(L, I)	I is the list of maximal intervals produced by the intersection of the lists of maximal intervals of list L
relative_complement_all(I', L, I)	I is the list of maximal intervals produced by the relative complement of the list of maximal intervals I' with respect to every list of maximal intervals of list L

We represent instantaneous SDE and CE by means of happensAt, while durative SDE and CE are represented as fluents. The task generally is to compute, for every durative CE of interest, the maximal intervals for which that CE holds.

Next we give a few example representations of CE definitions from CTM. The city transport officials are interested in computing, for instance, the intervals during which a vehicle is (non-)punctual. This may be achieved in RTEC as follows:

$$\text{initially}(punctuality(\_,\_) = punctual) \tag{1}$$

$$\begin{aligned}&\text{initiatedAt}(punctuality(Id, VT) = punctual,\ T) \leftarrow \\ &\quad \text{happensAt}(enter\_stop(Id, VT, Stop, scheduled),\ \_), \\ &\quad \text{happensAt}(leave\_stop(Id, VT, Stop, scheduled),\ T)\end{aligned} \tag{2}$$

$$\begin{aligned}&\text{initiatedAt}(punctuality(Id, VT) = punctual,\ T) \leftarrow \\ &\quad \text{happensAt}(enter\_stop(Id, VT, Stop, early),\ \_), \\ &\quad \text{happensAt}(leave\_stop(Id, VT, Stop, scheduled),\ T)\end{aligned} \tag{3}$$

$$\begin{aligned}&\text{initiatedAt}(punctuality(Id, VT) = non\_punctual,\ T) \leftarrow \\ &\quad \text{happensAt}(enter\_stop(Id, VT, \_, late),\ T)\end{aligned} \tag{4}$$

$$\begin{aligned}&\text{initiatedAt}(punctuality(Id, VT) = non\_punctual,\ T) \leftarrow \\ &\quad \text{happensAt}(leave\_stop(Id, VT, \_, early),\ T)\end{aligned} \tag{5}$$

$$\begin{aligned}&\text{initiatedAt}(punctuality(Id, VT) = non\_punctual,\ T) \leftarrow \\ &\quad \text{happensAt}(leave\_stop(Id, VT, \_, late),\ T)\end{aligned} \tag{6}$$

enter_stop and *leave_stop* are instantaneous SDE,

determined from sensor data and a database of timetable information. *Id* represents the id of a vehicle, *VT* represents the type of a vehicle (bus or tram), *Stop* is the code of a stop, and '_' is an 'anonymous' Prolog variable. Initially, every vehicle is punctual. Thereafter *punctuality* is affected by the *enter_stop* and *leave_stop* events. A vehicle is said to be punctual if it arrives at a stop on or before the scheduled time, and leaves the stop at the scheduled time. A vehicle is said to be non-punctual if it arrives at a stop after the scheduled time, or leaves the stop before or after the scheduled time. Computing the maximal intervals during which a vehicle is continuously (non-)punctual is achieved by computing the maximal intervals of *punctuality* using the built-in holdsFor predicate. RTEC provides a number of shorthand constructs to make the writing of initiatedAt rules more concise; we omit the details to save space.

Transport officials are also interested in recognising punctuality *change*. Consider the following CE definition:

$$\begin{aligned}&\mathsf{happensAt}(\mathit{punctuality\_change}(\mathit{Id}, \mathit{VT}, \mathit{Value}),\ T) \leftarrow\\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VT}) = \mathit{Value},\ I),\\ &\quad (T, \_) \in I,\\ &\quad T \neq 0\end{aligned} \tag{7}$$

This rule uses holdsFor to compute the maximal intervals for which a vehicle is continuously (non-)punctual. Punctuality changes at the first time-point of each of these intervals—see the penultimate condition of rule (7). There are other, equivalent ways to express this definition but since punctuality intervals are to be computed anyway, this method is convenient.

Briefly, to compute the maximal intervals during which a fluent F has value V continuously, that is, to compute $\mathsf{holdsFor}(F = V,\ I)$, we find all time-points T_s at which $F = V$ is initiated, and then, for each T_s, we compute the first time-point T_f after T_s at which $F = V$ is terminated. The time-points at which $F = V$ is initiated are computed with the use of initiatedAt rules. The time-points at which $F = V$ is terminated are computed with the use of broken:

$$\begin{aligned}&\mathsf{broken}(F = V,\ T_f) \leftarrow\\ &\quad \mathsf{terminatedAt}(F = V,\ T_f)\end{aligned} \tag{8}$$

$$\begin{aligned}&\mathsf{broken}(F = V_1,\ T_s) \leftarrow\\ &\quad \mathsf{initiatedAt}(F = V_2,\ T_s),\\ &\quad V_1 \neq V_2\end{aligned} \tag{9}$$

According to rule (9), if $F = V_2$ is initiated at T_s then effectively $F = V_1$ is terminated at time T_s, for all other possible values V_1 of F. Rule (9) ensures, therefore, that a fluent cannot have more than one value at any time. The RTEC implementation stores holdsFor intervals as they are computed for any given fluent F: thereafter intervals for F are retrieved from the computer memory without the need for re-computation.[2]

In addition to the domain-independent definition of holdsFor, an event description may include domain-dependent holdsFor rules, in particular to define a CE in terms of SDE and other CE. Such rules typically use interval manipulation constructs. RTEC supports three such constructs: union_all, intersect_all and relative_complement_all (see Table 1). Given a list L of maximal intervals, $\mathsf{union\_all}(L,\ I)$ computes the list I of maximal intervals corresponding to the union of the maximal intervals of L. Consider the following examples:

$$\mathsf{union\_all}([[(5,20),(26,30)],[],[(28,35)]],\quad [(5,20),(26,35)])$$

$$\begin{aligned}&\mathsf{union\_all}([[(5,20),(26,30)],[(1,4),(21,26)]],\\ &\qquad [(1,4),(5,20),(21,30)])\end{aligned}$$

A term of the form (T_s, T_e) represents the closed-open interval $[T_s, T_e)$. The implementation of all interval manipulation constructs, including union_all, is available with the code of RTEC.

$\mathsf{intersect\_all}(L,\ I)$ computes the list of maximal intervals I such that I is the intersection of the lists of maximal intervals of list L. Consider the following examples:

$$\mathsf{intersect\_all}([[(5,20),(26,30)],[(28,35)]],\quad [(28,30)])$$

$$\mathsf{intersect\_all}([[(5,20),(26,30)],[(1,4),(21,26),(30,40)]],\quad [])$$

$\mathsf{relative\_complement\_all}(I',\ L,\ I)$ computes the list of maximal intervals I such that I is the relative complement of the list of maximal intervals I' with respect to the maximal intervals of list L. Below are two examples:

$$\begin{aligned}&\mathsf{relative\_complement\_all}([(5,20),(26,50)],\\ &\qquad [[(1,4),(18,22)],[(28,35)]],\\ &\qquad [(5,18),(26,28),(35,50)])\end{aligned}$$

$$\begin{aligned}&\mathsf{relative\_complement\_all}([(5,20),(26,50),(60,70)],\\ &\qquad [[(1,4),(55,65)],[],[(52,80)]],\\ &\qquad [(5,20),(26,50)])\end{aligned}$$

Three example domain-dependent holdsFor rules using the interval manipulation constructs of RTEC are the following:

$$\begin{aligned}&\mathsf{holdsFor}(\mathit{driving\_quality}(\mathit{Id}, \mathit{VT}) = \mathit{high},\ I) \leftarrow\\ &\quad \mathsf{holdsFor}(\mathit{driving\_style}(\mathit{Id}, \mathit{VT}) = \mathit{uncomfortable}, I'),\\ &\quad \mathsf{holdsFor}(\mathit{driving\_style}(\mathit{Id}, \mathit{VT}) = \mathit{unsafe},\ I''),\\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VT}) = \mathit{punctual},\ I'''),\\ &\quad \mathsf{relative\_complement\_all}(I''',\ [I', I''],\ I)\end{aligned} \tag{10}$$

$$\begin{aligned}&\mathsf{holdsFor}(\mathit{driving\_quality}(\mathit{Id}, \mathit{VT}) = \mathit{medium},\ I) \leftarrow\\ &\quad \mathsf{holdsFor}(\mathit{driving\_style}(\mathit{Id}, \mathit{VT}) = \mathit{uncomfortable}, I'),\\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VT}) = \mathit{punctual},\ I''),\\ &\quad \mathsf{intersect\_all}([I', I''],\ I)\end{aligned} \tag{11}$$

$$\begin{aligned}&\mathsf{holdsFor}(\mathit{driving\_quality}(\mathit{Id}, \mathit{VT}) = \mathit{low},\ I) \leftarrow\\ &\quad \mathsf{holdsFor}(\mathit{driving\_style}(\mathit{Id}, \mathit{VT}) = \mathit{unsafe},\ I'),\\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VT}) = \mathit{non\_punctual},\ I''),\\ &\quad \mathsf{union\_all}([I', I''],\ I)\end{aligned} \tag{12}$$

Recall that *punctuality* was defined by rules (1)–(6). The definition of the *driving_style* CE is omitted to save space. High quality driving is recognised when a vehicle is punctual and the driving style is neither unsafe nor uncomfortable. Medium quality driving is recognised when the driving style is uncomfortable and the vehicle is punctual. Low quality driving is recognised when the driving style is unsafe or the vehicle is non-punctual. Again, RTEC provides some higher-level constructs to make such holdsFor specifications more readable and more concise. For example, rules (10)–(12)

[2]In Artificial Intelligence and other works on EC, this is often referred to as a form of 'caching'. We will avoid the use of this term, however, in case of possible confusion with other uses of the term.

can be written in the form:

$$\begin{aligned}
&driving_quality(Id, VT) = high \text{ iff}\\
&\quad driving_style(Id, VT) \neq uncomfortable,\\
&\quad driving_style(Id, VT) \neq unsafe,\\
&\quad punctuality(Id, VT) = punctual\\
&driving_quality(Id, VT) = medium \text{ iff}\\
&\quad driving_style(Id, VT) = uncomfortable,\\
&\quad punctuality(Id, VT) = punctual\\
&driving_quality(Id, VT) = low \text{ iff}\\
&\quad driving_style(Id, VT) = unsafe \text{ or}\\
&\quad punctuality(Id, VT) = non_punctual
\end{aligned}$$

Further details are omitted here.

The use of interval manipulation constructs leads to a concise definition of the CE concerning driving quality. In the absence of these constructs, one would have to adopt the traditional style of EC representation, that is, identify all possible conditions in which $driving\_quality(Id, VT) = high$ (respectively, *medium*, *low*) is initiated, in all combinations, all conditions in which this CE is terminated, and then use the domain-independent holdsFor predicate to compute the maximal intervals of the CE. Such a formalisation is much more complex and lower-level than the representation using interval manipulation as in rules (10), (11) and (12). In general, the interval manipulation constructs of RTEC may significantly simplify the definitions of durative CE. With the use of union_all, for example, we are able to develop succinct representations of most CE in the CTM application. The interval manipulation constructs can also lead to much more efficient CE recognition.

Fluents defined in terms of initiatedAt and terminatedAt rules, and whose maximal intervals are computed by means of the domain-independent holdsFor rules, such as *punctuality*, are called *simple*. Fluents defined in terms of domain-dependent holdsFor rules, such as *driving_quality*, or domain-dependent holdsAt rules (not shown here), are called *statically determined*.

4. RUN-TIME EVENT RECOGNITION

Typically, CE recognition has to be efficient enough to support real-time decision-making, and scale to very large numbers of SDE. These SDE may not necessarily arrive at the CE recognition system in a timely manner, that is, there may be a (variable) delay between the time at which SDE take place and the time at which they arrive at the CE recognition system (see [25] for a further discussion). Moreover, SDE may be revised, or even completely discarded in the future. Consider, for example, the case where the parameters of a SDE were originally computed erroneously and are subsequently revised, or the retraction of a SDE that was reported by mistake, and the mistake was realised later [2]. Note that SDE revision is not performed by the CE recognition system, but by the underlying SDE detection system. The effects of SDE revision are computed by the CE recognition system, provided that the latter supports such functionality.

RTEC performs run-time CE recognition by querying, computing and storing the maximal intervals of fluents and the time-points in which events occur. CE recognition takes place at specified query times $Q_1, Q_2, \ldots$. At each query time Q_i only the SDE that fall within a specified interval—the 'working memory' or 'window' (*WM*)—are taken into consideration: all SDE that took place before or on $Q_i - WM$ are discarded. This is to make the cost of CE recognition dependent only on the size of *WM* and not on the complete SDE history. As a consequence, of course, 'windowing' will potentially change the answer to some queries. Some of the stored sub-computations may have to be checked and possibly recomputed. Much of the detail of the RTEC algorithms is concerned with this requirement.

The size of *WM*, as well as the temporal distance between two consecutive query times—the 'step' ($Q_i - Q_{i-1}$)—is chosen by the user. Consider the following cases:

- $WM < Q_i - Q_{i-1}$, that is, *WM* is smaller than the step. In this case, the effects of the SDE that took place in $(Q_{i-1}, Q_i - WM]$ will be lost.
- $WM = Q_i - Q_{i-1}$. In this case, no information will be lost, *provided that* all SDE arrive at RTEC in a timely manner, and there is no SDE revision. If SDE do not arrive in a timely manner, then the effects of SDE that took place before Q_i but arrived after Q_i will be lost. Furthermore, if SDE are revised, the effects of the revision of SDE that took place before Q_i and were revised after Q_i will be lost.
- $WM > Q_i - Q_{i-1}$. In the common case that SDE arrive at RTEC with delays, or there is SDE revision, it is preferable to make *WM* longer than the step. In this way, it will be possible to compute, at Q_i, the effects of SDE that took place in $(Q_i - WM, Q_{i-1}]$, but arrived at RTEC after Q_{i-1}. Moreover, it will be possible to compute, at Q_i, the effects of the revision of SDE that took place in $(Q_i - WM, Q_{i-1}]$ and were revised after Q_{i-1}.

(*WM* is also called 'tumble window' [8] when $WM \leq Q_i - Q_{i-1}$ and 'pane window' when $WM > Q_i - Q_{i-1}$.) Note that even when $WM > Q_i - Q_{i-1}$ information may be lost. The effects of SDE that took place before or on $Q_i - WM$ and arrived after Q_{i-1} are lost. Similarly, the effects of the revision of SDE that took place before or on $Q_i - WM$ and were revised after Q_{i-1} are lost. To reduce the possibility of losing information, one may increase the size of *WM*; in this case, however, recognition efficiency will decrease. In what follows we give an illustrative example and a detailed account of how the 'windowing' works in CE recognition.

4.1 Illustrative Example

Figure 1 illustrates the windowing algorithm of RTEC. In this example we have $WM > Q_i - Q_{i-1}$. To avoid clutter, Figure 1 shows streams of only five SDE. These are displayed below *WM*, with dots for instantaneous SDE and lines for durative SDE. In this example, we are interested in recognising just two CE:

- CE_{simple}, represented as a simple fluent. The starting and ending points as well as the maximal intervals of CE_{simple} are displayed directly above *WM* in Figure 1.
- CE_{sd}, represented as a statically determined fluent. For illustration purposes, we define the maximal intervals of CE_{sd} to be the union of the maximal intervals of the two durative SDE displayed in Figure 1. The maximal intervals of CE_{sd} are displayed above the CE_{simple} intervals in Figure 1.

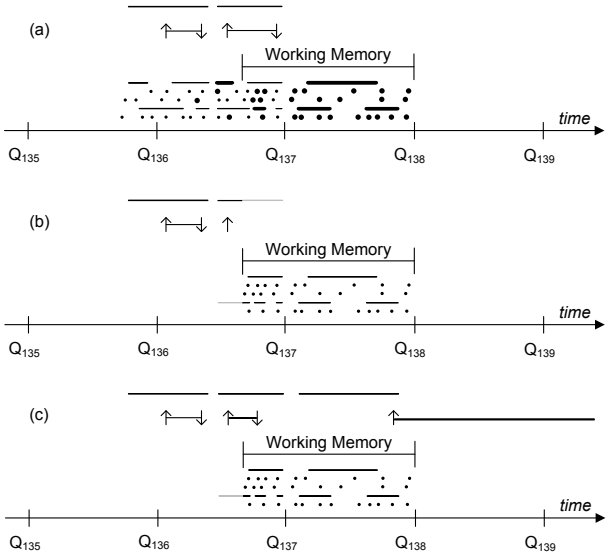

Figure 1: Windowing in RTEC.

To simplify the illustration, we assume that both CE_{simple} and CE_{sd} are defined only in terms of SDE, that is, they are not defined in terms of other CE.

Figure 1 shows the steps that are followed in order to perform CE recognition at an arbitrary query time, say Q_{138}. This figure shows the SDE available at Q_{138}. All SDE that took place before or on $Q_{137}-WM$ were retracted at Q_{137}. Between Q_{137} and Q_{138} several SDE arrived at the system, some of which took place before Q_{137}. For illustration purposes, these are represented by thick lines and dots in Figure 1(a). The effects of SDE that arrived between Q_{137} and Q_{138} and took place before or on $Q_{138}-WM$ are lost. Figure 1(b) shows that all SDE that took place before or on $Q_{138}-WM$ are discarded. All SDE that took place in $(Q_{138}-WM, Q_{138}]$ will be considered in the CE recognition process at Q_{138}. The interval of each durative SDE that started before $Q_{138}-WM$ and ended after that time is partly retracted: RTEC retracts the sub-interval up to and including $Q_{138}-WM$. Figure 1(b) shows the interval of a SDE that is partly retracted in this way: the discarded sub-interval is grayed out.

Apart from discarding all SDE as described above, we also discard at Q_i all CE intervals in $(Q_i-WM, Q_i]$. These intervals might not hold given the SDE that arrived or were revised after Q_{i-1} since it is possible that some of these SDE took place in $(Q_i-WM, Q_{i-1}]$. Determining which CE intervals should be partly or completely retracted can be computationally very expensive. See Section 6 for a discussion. Therefore, we choose to discard all CE intervals in $(Q_i-WM, Q_i]$ and compute everything from scratch.

RTEC does not manipulate the CE intervals that have ended before or on Q_i-WM. Depending on the user requirements, these intervals may be stored in a database for retrospective inspection of the activities of a system. (To avoid clutter, Figure 1(a) shows only the CE intervals computed at Q_{137}.)

Figure 1(b) shows that the last interval of CE_{sd} at Q_{137} was partly retracted when CE recognition started at Q_{138}. This happened because the starting point of the interval was before $Q_{138}-WM$ while its ending point was after that time. The part of the interval that was retracted is grayed out in Figure 1(b).

The maximal intervals of statically determined CE are computed by evaluating domain-dependent holdsFor rules. In the example, we calculate the maximal intervals of CE_{sd} by computing the union of the lists of maximal intervals of the two durative SDE shown in Figure 1. Note that, at Q_{138}, only the SDE intervals in $(Q_{138}-WM, Q_{138}]$ are considered in the computation of the CE_{sd} intervals.

It may happen that the first interval of a statically determined CE computed at Q_i starts immediately after Q_i-WM. Moreover, it is possible that there is an interval of such a CE that ends on Q_i-WM. In the example, the second interval of CE_{sd} was partly retracted and, as a result, ended on $Q_{138}-WM$. In order to deal with such cases, RTEC amalgamates the last interval taking place before or on Q_i-WM with the first interval taking place in $(Q_i-WM, Q_i]$. The result of this process for CE_{sd} at Q_{138} is exactly the same interval as the second interval of CE_{sd} that was computed at Q_{137}: compare the second interval of CE_{sd} shown in Figure 1(c) with the second interval of this CE shown in Figure 1(a). In other words, in this example, the SDE that arrived after Q_{137} and took place in $(Q_{138}-WM, Q_{137}]$ did not affect the intervals of CE_{sd}. Had CE_{sd} been defined in a different way, say as the *intersection* of the lists of maximal intervals of the two durative SDE in Figure 1, then the intervals of CE_{sd} would have changed in $(Q_{138}-WM, Q_{137}]$.

Figure 1 also shows the way in which the intervals of the simple fluent CE_{simple} are computed at Q_{138}. Arrows facing upwards (downwards) denote the starting (ending) points of CE_{sd} intervals. First, the last interval of CE_{simple} is completely retracted, and only the starting point of that interval is kept. See Figure 1(b). This interval is retracted because it starts before $Q_{138}-WM$ and ends after that time. It is simpler to retract this interval completely and reconstruct it with the use of its starting point and the domain-independent holdsFor rules, rather than keeping the sub-interval that takes place before $Q_{138}-WM$, and possibly amalgamating it later with another interval, as we do for CE expressed as statically determined fluents.

If the last, or any other interval of CE_{simple} that was computed at Q_{137}, had started after $Q_{138}-WM$, then both the interval and its starting point would have been discarded when the CE recognition process commenced at Q_{138}.

All ending (and starting) points after $Q_{138}-WM$, computed at Q_{137}, are also discarded.

The second step we take concerning CE_{simple} at Q_{138} is to calculate its starting and ending points. We compute these points by evaluating initiatedAt and broken rules. To evaluate the conditions of such rules we only consider SDE that took place in $(Q_{138}-WM, Q_{138}]$. Figure 1(c) shows the starting and ending points of CE_{simple} in $(Q_{138}-WM, Q_{138}]$. Note that the last ending point of CE_{simple} that was computed at Q_{137} was invalidated in the light of the new SDE that became available at Q_{138} (compare Figures 1(c)–(a)). Moreover, another ending point was computed at an earlier time.

The final step we take in order to recognise CE_{simple} at Q_{138} is to use the domain-independent holdsFor predicate to calculate the maximal intervals of CE_{simple} given its starting and ending points. The second interval of CE_{simple} became shorter than that computed at Q_{137} (compare Figures 1(c)–(a)), while the last interval of CE_{simple} is open: given the

SDE available at Q_{138}, we say that CE_{simple} holds *since* time t, where t is the last starting point of CE_{simple}.

The example presented above illustrates the possibility that, when SDE arrive with a variable delay, CE intervals computed at an earlier query time may be, partly or completely, retracted at the current or a future query time. (And similarly if SDE are revised.) Depending on the requirements of the application, RTEC may report to the user:

- CE as soon as they are recognised, even if the intervals of these CE may be partly or completely retracted in the future.
- CE whose intervals may be partly, but not completely, retracted in the future, that is, CE whose intervals start before or on $Q_{i+1}-WM$ and end after that time.
- only CE whose intervals will not be even partly retracted in the future, that is, CE whose intervals end before or on $Q_{i+1}-WM$.

4.2 RTEC Operation

In this section we first present the compilation stage of RTEC and then discuss the way RTEC operates at run-time. The run-time activities of RTEC consist of the mechanisms for discarding 'old' SDE and the CE recognition process itself.

4.2.1 Compilation

Before the commencement of run-time activities, RTEC compiles the CE definitions into a format that allows for more efficient CE recognition. This is a process transparent to the user. Any shorthand abbreviations are also expanded at this stage. The aim of the transformation is to eliminate the number of unsuccessful evaluations of holdsFor, and to introduce additional indexing information. In particular, all holdsFor atoms appearing in a CE definition are rewritten using specialised predicates, depending on whether they appear in the head or the body of a rule, and whether they concern a simple or a statically determined fluent. Specifically, holdsFor atoms appearing in the head of a domain-dependent rule, that is, a rule for computing the maximal intervals of statically determined fluents, are rewritten using the predicate holdsForSDFluent. holdsFor atoms appearing in the body of a rule are translated into holdsForRecognisedSimpleFluent atoms or holdsForRecognisedSDFluent atoms according to whether they concern simple fluents or statically determined ones.

RTEC computes CE intervals in a bottom-up manner: it first recognises 'level-1' CE, that is, CE defined only in terms of SDE, then it recognises 'level-2' CE, that is, CE defined in terms of at least one level-1 CE and a (possibly empty) set of SDE, then it recognises 'level-3' CE, that is, CE defined in terms of at least one level-2 CE and a (possibly empty) set of SDE and level-1 CE, and so on. In other words, when recognising a CE C all CE C_j appearing in the body of each rule defining C will already have been recognised and their intervals stored. holdsForRecognisedSDFluent and holdsForRecognisedSimpleFluent are defined as follows:

$$\text{holdsForRecognisedSimpleFluent}(Index,\ F=V,\ I) \leftarrow \text{simpleFList}(Index,\ F=V,\ I,\ \_) \quad (13)$$

$$\text{holdsForRecognisedSDFluent}(Index,\ F=V,\ I) \leftarrow \text{sdFList}(Index,\ F=V,\ I,\ \_) \quad (14)$$

simpleFList and sdFList are the predicates used to store the 'cache' of computed intervals in Prolog's dynamic working memory. The third argument in each case stores the list of intervals starting in $(Q_i-WM, Q_i]$ for which simple (respectively, statically determined) fluent F has value V. The first argument is an index that allows for the fast retrieval of stored intervals for a given fluent even in the presence of very large numbers of fluents. (We index events in a similar manner.) This is very important in large-scale applications. We show the effects in the experiments presented in Section 5. RTEC adds the index at the compilation stage in the transformation to holdsForRecognisedSDFluent and holdsForRecognisedSimpleFluent. The choice of index for a fluent is declared by the user. In the CTM application, for example, the index of all CE fluents is the vehicle id, since all queries tend to be about specific vehicles. More details on how these specialised holdsFor predicates are used with simpleFList and sdFList in the CE recognition process will be given in Section 4.2.3.

4.2.2 Forget Mechanism

At each query time Q_i, RTEC first discards—'forgets'—all SDE that end before or on Q_i-WM. For each SDE available at Q_i, RTEC:

- Completely retracts the SDE if the interval attached to it ends before or on Q_i-WM.
- Partly retracts the interval of the SDE if it starts before or on Q_i-WM and ends after that time. For each SDE type there can be at most one such interval. More precisely, RTEC retracts the SDE interval $(Start, End)$ and asserts the interval (Q_i-WM, End).

4.2.3 Composite Event Recognition

After 'forgetting' SDE, RTEC recognises the CE of interest, that is, computes and stores the intervals of each such CE fluent. At the end of CE recognition at each query time Q_i, all computed fluent intervals are stored as simpleFList and sdFList assertions in Prolog memory, indexed by fluent as described above. For example, I in $\text{sdFList}(Index, CE_{sd}, I, PE)$ represents the intervals of statically determined fluent CE_{sd} starting in $(Q_i-WM, Q_i]$, sorted in temporal order. PE stores the interval, if any, ending at Q_i-WM. When the user queries the maximal intervals of CE_{sd}, RTEC amalgamates PE with the intervals in I, producing a list of maximal intervals ending in $[Q_i-WM, Q_i]$ and, possibly, an open interval starting in $[Q_i-WM, Q_i]$. In what follows, we present how RTEC computes and stores the maximal intervals of fluents at each Q_i. Computing and storing the time-points of instantaneous events is simpler and so we do not present the details here to save space.

Listing 1 shows the pseudo-code of recogniseSDFluent, that is, the procedure for computing and storing the intervals of statically determined fluents. First, RTEC retrieves the maximal intervals of a statically determined fluent CE_{sd} computed at Q_{i-1} and checks if there is such an interval that starts before or on Q_i-WM and ends after or at that time. If there is such an interval then the sub-interval, if any, after Q_i-WM will be discarded. As already mentioned, we compute all CE intervals from scratch in $(Q_i-WM, Q_i]$. To determine if there is an interval of CE_{sd} that starts before or on Q_i-WM and ends after or at that time, RTEC looks through the intervals stored in sdFList. See Listing 1. At

Listing 1 recogniseSDFluent(CE_{sd}, $Index$, $Q_i - WM$)

```
{partly discard the statically determined fluent interval,
if any, that starts before or on Q_i − WM and ends after}
{terms (T_s, T_e) in RTEC represent intervals [T_s, T_e)}
sdFList(Index, CE_sd, OldI, OldPE)
amalgamate(OldPE, OldI, OldList)
if OldList ≠ [] then
  if Start, End : (Start, End) ∈ OldList ∧
     End > Q_i − WM ∧ Start ≤ Q_i − WM then
       PE := [(Start, Q_i − WM + 1)]
  else
       PE := []
  end if
end if
{compute statically determined fluent intervals}
holdsForSDFluent(CE_sd, I)
retract(sdFList(Index, CE_sd, OldI, OldPE))
assert(sdFList(Index, CE_sd, I, PE))
```

Listing 2 recogniseSimpleFluent(CE_{simple}, $Index$, $Q_i - WM$)

```
{keep the starting point of the simple fluent interval, if
any, that starts before or on Q_i − WM and ends after}
simpleFList(Index, CE_simple, OldI, OldPE)
amalgamate(OldPE, OldI, OldList)
if OldList ≠ [] then
  if Start, End : (Start, End) ∈ OldList ∧
     End > Q_i − WM + 1 ∧ Start ≤ Q_i − WM then
       OldSPoint := [Start]
  else
       OldSPoint := []
  end if
end if
{compute simple fluent intervals}
computeStartingPoints(CE_simple, NewSPoints)
holdsForSimpleFluent(OldSPoint ∪ NewSPoints, CE_simple, I')
computeSimpleFList(I', Q_i − WM, I, PE)
retract(simpleFList(Index, CE_simple, OldI, OldPE))
assert(simpleFList(Index, CE_simple, I, PE))
```

this point, *OldI* represents the intervals of CE_{sd} computed at Q_{i-1}. These intervals are temporally sorted and start in $(Q_{i-1} - WM, Q_{i-1}]$. *OldPE* stores the interval, if any, ending at $Q_{i-1} - WM$. RTEC amalgamates *OldPE* with the intervals in *OldI*, producing *OldList*. RTEC goes through the maximal intervals in *OldList* until an interval that ends after or on $Q_i - WM$ is found. If such an interval is found, then the sub-interval before or on $Q_i - WM$ is stored. See *PE* in Listing 1.

At the second step of recogniseSDFluent, RTEC evaluates holdsForSDFluent rules to compute the intervals of CE_{sd}. Recall that, at the compilation stage, RTEC transforms all holdsFor rules concerning statically determined fluents into holdsForSDFluent rules. The intervals of CE_{sd} computed at the previous query time Q_{i-1} are not used. The computed list of intervals I of CE_{sd}, along with *PE*, are stored in sdFList, replacing the intervals computed at Q_{i-1}. (As mentioned above, to answer user queries, RTEC amalgamates *PE* with the intervals in I.)

Listing 2 shows the pseudo-code of recogniseSimpleFluent, that is, the procedure for computing and storing simple fluent intervals. Similarly to recogniseSDFluent, this procedure has two main parts. First, RTEC checks if there is a maximal interval of the fluent CE_{simple} that starts before or on $Q_i - WM$ and ends after that time. This is determined by looking through the intervals stored in simpleFList. If there is such interval then it will be discarded, while its starting point will be kept—see *OldSPoint* in Listing 2.

At the second step of recogniseSimpleFluent, RTEC computes the starting points of CE_{simple}, without considering the starting points calculated at Q_{i-1}. The computed starting points, along with *OldSPoint*, are given as input to holdsForSimpleFluent, into which holdsFor calls computing the maximal intervals of simple fluents were translated at compile time. This predicate is defined as follows:

$$\text{holdsForSimpleFluent}([],\ \_,\ []) \tag{15}$$

$$\begin{aligned}&\text{holdsForSimpleFluent}(SPoints,\ CE_{simple},\ I) \leftarrow\\ &\quad SPoints \neq [],\\ &\quad \text{computeEndingPoints}(CE_{simple},\ EPoints),\\ &\quad \text{makeIntervalsFromSEPoints}(SPoints,\ EPoints,\ I)\end{aligned} \tag{16}$$

If the list of starting points is empty (first argument) then the empty list of intervals is returned (see rule (15)). Otherwise, holdsForSimpleFluent computes the ending points of the simple fluent, without considering the ending points calculated at Q_{i-1}, and then uses makeIntervalsFromSEPoints to compute its maximal intervals given its starting and ending points (see rule (16)).

The first interval computed by holdsForSimpleFluent could start at $Q_i - WM$ or earlier, as there may be a starting point computed at the first step of recogniseSimpleFluent. computeSimpleFList sets the part of the first interval up to or on $Q_i - WM$ to *PE*. The remaining sub-interval, along with the remaining maximal intervals, are recorded in I. I and *PE* are stored in simpleFList, replacing the intervals computed at Q_{i-1}.

4.3 Complexity

In this section, we analyse the complexity of the 'forget' mechanism and the computation of statically determined fluent intervals. Due to space limitations, it is not possible to present a complete account of the complexity of RTEC.

In the analysis below, $m(S, E)$ denotes the number of time-points in the interval $(S, E]$—we assume discrete time. $m(S, E)/2$ is thus the maximum number of maximal intervals in $(S, E]$. The number of time-points in WM, $m(Q_i - WM, Q_i)$, is denoted in short by m_{WM}. The maximum number of maximal intervals in WM is $m_{WM}/2$.

4.3.1 Forget Mechanism

At each query time Q_i, RTEC first 'forgets' all available SDE ending before or on $Q_i - WM$. If the list of available SDE is temporally sorted then RTEC stops processing SDE as soon as it finds the first one that starts after $Q_i - WM$. In the common case that SDE arrive with a variable delay, RTEC goes through the complete list of SDE available at Q_i. In the worst case, all SDE that took place in $(0, Q_i]$ arrive between Q_{i-1} and Q_i. The worst-case cost of the 'forget' mechanism is thus

$$\mathcal{O}(\, n\,(m(0, Q_i) + m(0, Q_i - WM)\,)) \tag{17}$$

where n denotes the number of SDE types. This is the cost of going through the SDE in $(0, Q_i]$ and retracting those in $(0, Q_i - WM]$. This situation may occur at most once

since all SDE 'forgotten' at Q_i are not available after Q_i. In practice, the cost of the 'forget' mechanism is bounded by approximately

$$n\,(\,m(Q_{i-1}-WM, Q_i) + m(Q_{i-1}-WM, Q_i-WM)\,) \quad (18)$$

that is, the SDE that took place before or on $Q_{i-1}-WM$ are (typically) retracted at Q_{i-1} and are not available at Q_i.

4.3.2 Statically Determined Fluents

At the first step of recogniseSDFluent, RTEC searches the maximal intervals of the fluent in question ending in $[Q_{i-1}-WM, Q_{i-1}]$ and, possibly, an open interval starting in $[Q_{i-1}-WM, Q_{i-1}]$. The worst-case cost of this step is

$$\mathcal{O}\left(\frac{m_{WM}}{2}+1\right) \quad (19)$$

In practice, the number of maximal intervals of a fluent ending in $[Q_{i-1}-WM, Q_{i-1}]$ is considerably smaller than the maximum number of maximal intervals in WM.

At the second step of recogniseSDFluent, RTEC evaluates a holdsForSDFluent rule. The cost of evaluating such a rule is limited by the sum of the cost of computing the intervals of the fluents appearing in the body of the rule and the cost of any interval manipulation operations. A fluent appearing in the body of a holdsForSDFluent rule represents a SDE or a CE. In either case, RTEC simply retrieves the fluent intervals from the computer memory (this should not be confused with WM). RTEC performs recognition bottom-up and thus the intervals of all CE appearing in the body of a holdsForSDFluent rule are already calculated when evaluating this rule: RTEC need only retrieve the intervals stored in simpleFList and sdFList. The third arguments of simpleFList and sdFList record intervals starting in $(Q_i-WM, Q_i]$, sorted in temporal order. Moreover, SDE intervals start in $(Q_i-WM, Q_i]$ as earlier intervals have been retracted by the 'forget' mechanism, and they are temporally sorted because RTEC sorts the intervals of durative SDE used in the definitions of the CE we want to recognise. Each fluent in the body of a holdsForSDFluent rule, therefore, has at most $m_{WM}/2$ temporally sorted maximal intervals.

The cost of the interval manipulation constructs of RTEC is as follows. To compute the union of a list of lists of maximal intervals, RTEC recursively uses iset_union for calculating the union of two lists of maximal intervals. The cost of iset_union is limited by the sum of the sizes of the two lists, as this predicate operates under the assumption that each list of maximal intervals is sorted. Furthermore, the size of the output list of iset_union is limited by the sum of the sizes of the two lists, as, in the worst case, the intervals of the two input lists of iset_union are disjoint. Assuming x lists of maximal intervals of size y, the cost of union_all is bounded by:

$$\mathcal{O}(\overbrace{2y}^{1st\ \text{iset\_union}} + \overbrace{2y+y}^{2nd\ \text{iset\_union}} + \ldots + \overbrace{2y+y+\ldots+y}^{x-1th\ \text{iset\_union}}) = \\ = \mathcal{O}\left(y\left(\frac{x(x+1)}{2}-1\right)\right) \quad (20)$$

To compute the intersection of a list of lists of maximal intervals, RTEC recursively uses iset_intersection for calculating the intersection of two lists of maximal intervals. Like iset_union, the cost of iset_intersection is limited by the sum of the sizes of the two lists, as it operates under the assumption that each list of maximal intervals is sorted. The size of the output list of iset_intersection is bounded by the size of the longest input list. The cost of intersect_all is bounded by:

$$\mathcal{O}(\overbrace{2y}^{1st\ \text{iset\_intersection}} + \ldots + \overbrace{2y}^{x-1th\ \text{iset\_intersection}}) = \\ = \mathcal{O}(2y(x-1))$$

relative_complement_all(I', L, I) recursively uses iset_difference to compute the relative complement of the list of maximal intervals I' with respect to each list of maximal intervals of list L. The cost of iset_difference is limited by the sum of the sizes of the two input lists. Moreover, the size of the output list of iset_difference is limited by the sum of the sizes of the two lists. The cost of relative_complement_all, therefore, is the same as that of union_all.

Assuming that in the body of a holdsForSDFluent rule there are f fluents (SDE and CE)—in the worst case this is the number of the fluent types of the event description—and k interval manipulation constructs, the cost of evaluating such a rule is bounded by

$$\mathcal{O}\left(f + k\,\frac{m_{WM}}{2}\left(\frac{f(f+1)}{2}-1\right)\right) \quad (21)$$

This is the cost of retrieving f fluent intervals from the computer memory plus k times the cost of the most expensive interval manipulation construct (see formula (20)).

In practice, f and k are small, and the number of maximal intervals of a fluent starting in $(Q_i-WM, Q_i]$ is considerably smaller than $m_{WM}/2$.

5. EXPERIMENTAL RESULTS

We have evaluated RTEC experimentally on several example domains. Here we will present experiments on CTM. These experiments were performed on a computer with Intel i7 950@3.07GHz processors and 12GiB RAM, running Ubuntu Linux 11.04 and YAP Prolog 6.2.0. The number of processors varied by experiment, as described below. The real datasets (collected in November 2011 in Helsinki) include only a subset of the anticipated SDE types as some components detecting SDE were not functional. For that reason, in order to provide a more systematic and more stringent evaluation, we also performed experiments on artificially generated (synthetic) datasets as well as on real data. The synthetic datasets include the instantaneous SDE *enter_stop*, *leave_stop*, *passenger_density_change*, *temperature_change* and *noise_level_change*, and the durative SDE *abrupt_acceleration*, *abrupt_deceleration* and *sharp_turn*. Each synthetic SDE stream includes equal numbers of SDE types. In both cases—synthetic and real datasets—the SDE are not chronologically ordered. Given a synthetic or real SDE stream, RTEC recognises various CE including *punctuality*, *punctuality_change*, *driving_quality*, *driving_style*, *passenger_comfort*, *driver_comfort* and *passenger_satisfation*. These were specified by the end users.

Figure 2 shows a number of experimental results on synthetic datasets regarding CE recognition for a single vehicle. These were intended to test the effects of varying the size of WM and the tolerance of RTEC to irrelevant SDE. The figure shows the results of four sets of experiments. In the first, only 10% of the SDE concern the vehicle for which we perform CE recognition. In the second and the third, 30%

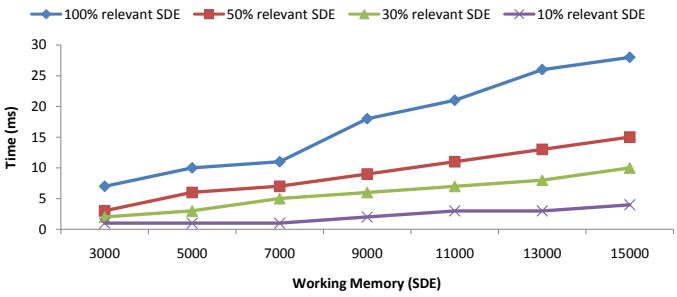

Figure 2: CE recognition for a single vehicle.

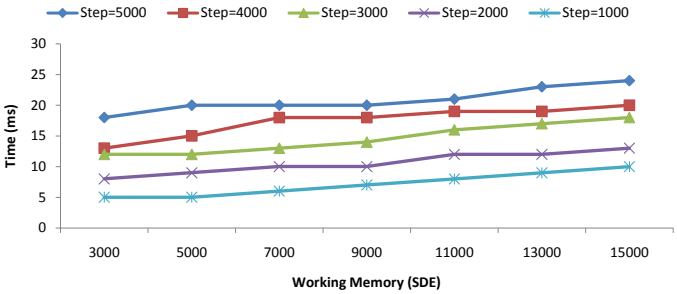

Figure 4: 'Forget' mechanism in CTM.

and 50% respectively of the SDE concern this vehicle. In the fourth case, all available SDE concern it. In every case, RTEC computes and stores the intervals of 20 CE types—this is the number of fluent and event types concerning an individual vehicle. We also varied the size of *WM*. Figure 2 shows results of experiments in which *WM* varies from 3000 to 15000 SDE. The times displayed in this figure show average CE recognition time in CPU milliseconds (ms).

In the present RTEC implementation, the indexing mechanism is very simple. (It merely exploits YAP Prolog's standard indexing on the functor of the first argument of the head of a clause). Nevertheless, as shown in Figure 2, the presence of irrelevant SDE affects recognition efficiency only very slightly. This is a very important feature of our approach as it means we do not have to rely on modules filtering SDE.

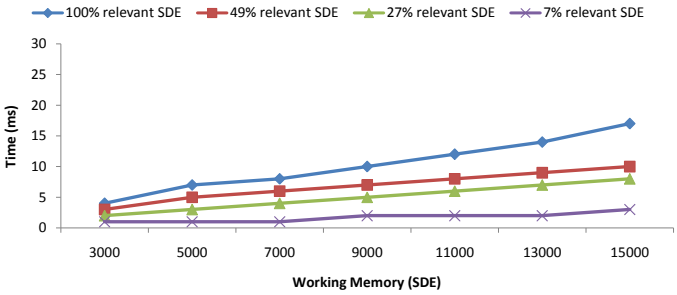

Figure 3: CE recognition for a single vehicle (real datasets).

For comparison, Figure 3 shows the results of the same experiments on real datasets. (Here the percentages of relevant SDE are determined by the data that were collected.) As explained earlier, the absence of several SDE types in the real datasets simplified the CE recognition process, which accounts for the apparent improvement in performance.

At each query time, RTEC first 'forgets' 'old' SDE and then performs CE recognition. The times shown in Figures 3 and 2 do not include the time required by the 'forget' mechanism. The cost of this mechanism depends on the size of *WM* as well as the size of the step between consecutive query times Q_{i-1} and Q_i. Figure 4 shows the average time of the 'forget' mechanism under varying *WM* and step sizes. When *WM* includes 7000 SDE and the step includes 3000 SDE, for example, the average time required by the 'forget' mechanism is 13 ms.

The results shown in Figure 4 concern synthetic datasets. The times achieved on real datasets are very similar and therefore omitted.

For a given step, the time required by the 'forget' mechanism mostly increases as *WM* increases. This is due to the fact that RTEC has to go through a larger list of SDE when deciding which ones to 'forget'. For example, when *WM* includes 5000 SDE and the step includes 2000 SDE, the 'forget' mechanism of RTEC has to go through 7000 SDE at each query time. If we set the size of *WM* to 9000 SDE and keep the same step, the 'forget' mechanism of RTEC will have to go through 11000 SDE at each query time. RTEC has to go through the complete list of SDE available at a query time, in order to decide which ones to 'forget', as the SDE streams in the CTM application are not necessarily temporally sorted.

For a given *WM*, the time required by the 'forget' mechanism increases as the step increases. Like the case of increasing *WM*, RTEC has to go through larger lists of SDE when deciding which ones to 'forget'. Unlike the case of increasing *WM*, RTEC 'forgets' a larger number of SDE. For example, when the step includes 3000 SDE, RTEC 'forgets' 3000 SDE, when the step includes 5000 SDE, RTEC 'forgets' 5000 SDE, and so on.

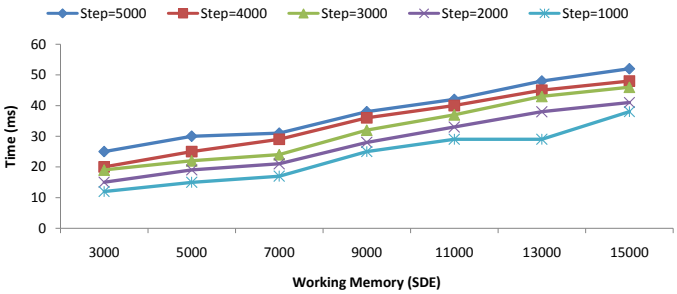

Figure 5: Total RTEC time: CE recognition for a single vehicle, 100% relevant SDE.

To compute the total time required by RTEC at each query time, one has to add the time required by the 'forget' mechanism to the time required for CE recognition. Figure 5 shows the time required by RTEC under varying *WM* and step sizes when all SDE are related to the vehicle for which we perform CE recognition. The times shown in this figure are produced by adding the times shown in Figure 4 and those corresponding to the '100% relevant SDE' line of

Figure 2. Note that the cost of the 'forget' mechanism is independent of how many SDE are related to the vehicle for which we perform CE recognition.

Most of the results presented in Figures 5 and 4 concern settings in which WM is larger than the step, that is, $WM > Q_i - Q_{i-1}$. There are two settings in which $WM = Q_i - Q_{i-1}$, and two settings in which $WM < Q_i - Q_{i-1}$. The sizes of WM and the step are chosen by the user (city transport officials, in this application). Due to the variable delay in SDE arrival in CTM, we expect that the user will choose a setting in which $WM > Q_i - Q_{i-1}$.

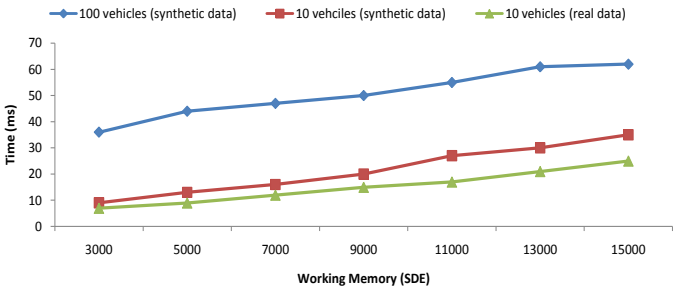

Figure 6: CE recognition for many vehicles (real and synthetic datasets).

Figure 6 shows experimental results regarding CE recognition for several vehicles. First, we perform CE recognition for 10 vehicles on real datasets and synthetic datasets. Second, we perform CE recognition for 100 vehicles on synthetic datasets. We do not have real datasets at the scale of 100 vehicles. In the first case, each vehicle is associated with 10% of the SDE, while in the second case each vehicle is associated with 1% of the SDE. The times shown in Figure 6 do not include the time required by the 'forget' mechanism. The cost of this mechanism is independent of the number of vehicles for which we perform CE recognition and is shown in Figure 4.

In these experiments, RTEC recognises and stores substantially greater numbers of CE than the number of CE recognised in the experiments presented earlier. In the first case (CE recognition for 10 vehicles), RTEC recognises 200 CE—20 CE are associated with each vehicle—while in the second case (CE recognition for 100 vehicles), RTEC recognises 2000 CE. Figure 6 shows that the substantial increase of CE hardly affects the efficiency of CE recognition per vehicle. For example, the average time required for CE recognition for a single vehicle in the presence of 200 CE—divide by 10 the times for '10 vehicles (synthetic data)' and '10 vehicles (real data)' in Figure 6—is almost the same as the time required for CE recognition for a single vehicle in the presence of 20 CE, as shown by the corresponding times for '10% relevant SDE' of Figure 2 (respectively Figure 3).

This result may seem surprising. One may have expected that evaluating rules (13) and (14), for example, would take longer and longer as the number of CE increases, because we would have to go through longer lists of CE in order to retrieve from the memory the computed intervals of any given CE. We avoid this in RTEC by indexing the CE. Thereby, the search for the intervals of a given CE becomes very efficient, even in the presence of a very large number of CE.

Figure 6 also shows that RTEC performs better in the real datasets than in the synthetic ones. As mentioned earlier, this is due to absence of a few SDE types in the real datasets.

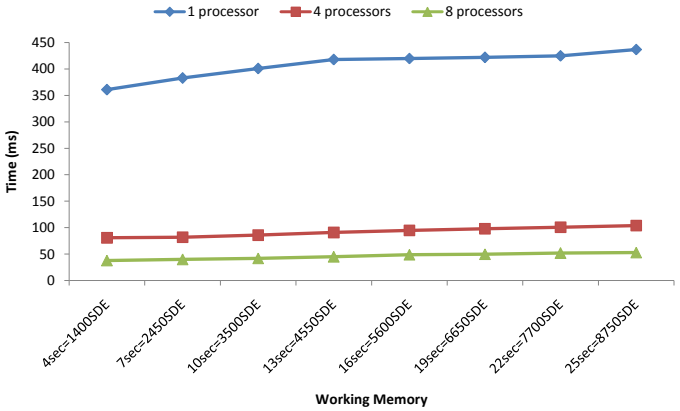

Figure 7: Total RTEC time: CE recognition during rush hour in Helsinki, step set to 1 sec = 350 SDE.

The last set of CTM experiments concerns CE recognition at rush hour in Helsinki. At most 1050 vehicles, that is, 80% of the total number of available vehicles, operate at the same time in Helsinki during rush hour. Due to the unavailability of real datasets at that scale, we simulated rush hour operations using synthetic datasets. It is estimated by the experts that no more than 350 SDE can be detected per second on the 1050 operating vehicles.[3] We were thus able to test RTEC under the maximum expected frequency of SDE.

Figure 7 presents the results of three sets of experiments. First, we used a single processor to perform CE recognition for all 1050 vehicles. In this case, the intervals of 21000 CE (1050 vehicles × 20 CE per vehicle) are computed and stored. Second, we used four processors in parallel. Each instance of RTEC running on a processor performed CE recognition for one quarter of all operating vehicles, that is, 263 vehicles, computing and storing the intervals of 5260 CE. Third, we used all eight processors of the computer in parallel. Each instance of RTEC running on a processor performed CE recognition for one eighth of all operating vehicles, that is, 132 vehicles, and computed and stored the intervals of 2640 CE.

In all sets of experiments the input was the same: SDE coming from *all* 1050 vehicles. In other words, there was no filtering of SDE data in these experiments to restrict the input relevant for each processor.

The times shown in Figure 7 include the time required by the 'forget' mechanism. The step is set to 1 sec (350 SDE), while WM ranges from 4 sec (1400 SDE) to 25 sec (8750 SDE). We found (in experiments not presented due to lack of space) that reducing the step size reduces recognition times very slightly. Given the current infrastructure in Helsinki, a 10 sec WM is sufficient, that is, a delay in the arrival of a SDE is expected to be less than 10 sec. Other CTM infrastructures may require different WM sizes.

Figure 7 shows that we can achieve a significant perfor-

[3]Personal communication with Mattersoft Ltd (`http://www.mattersoft.fi/en/index.html`).

mance gain by running RTEC in parallel on different processors. Such a gain is achieved without requiring SDE filtering.

In other application domains, SDE frequency may be higher than that presented above. According to the results of the use case survey of the Event Processing Technical Society (EPTS) [4], in most applications there are at most 1000 SDE per second. Our experimental evaluation showed that RTEC supports real-time reasoning in such applications. Consider, for example, Figure 5. Given a *WM* of 15000 SDE, which corresponds to a window of 15 sec in most applications according to the EPTS survey, recognition for a single vehicle is performed in less than 60 ms, when all SDE affect the CE we want to recognise. In the last set of experiments, we showed that in a *WM* of 8750 SDE, corresponding to a window of around 9 sec in most applications according to the EPTS survey, recognition for 1050 vehicles (21000 CE) is performed in about 50 ms. These results were achieved on a standard desktop computer.

6. RELATED WORK

One of the best-known recognition systems is the Chronicle Recognition System (CRS) [11]. CRS has proven efficient and scalable enough for various application domains. However, it is a purely temporal reasoning system and thus cannot be directly used for CE recognition in applications requiring any type of atemporal reasoning. In our approach to CE recognition, the availability of the full power of logic programming is one of the main attractions of employing RTEC as the temporal formalism. It allows CE definitions to include not only complex temporal constraints but also, when necessary, complex atemporal constraints. Moreover, it allows reasoning over CE definitions as well as reasoning over background knowledge. This is in contrast to various approaches, as pointed in [2, 3], such as [11, 16, 20, 7], that perform pattern matching over event streams, but lack the ability of (complex) reasoning over existing domain knowledge. An account of the benefits of logic programming over other approaches to CE recognition may be found in [21].

Logic programming approaches to CE recognition may be found in [24, 2, 3], for example. A distinguishing feature of our approach with respect to such lines of work concerns the fact that we use an EC dialect for temporal representation and reasoning. RTEC has built-in axioms for complex temporal representation, including the formalisation of inertia, which facilitate considerably the development of succinct CE definitions, and, therefore, code maintenance.

The Cached Event Calculus (CEC) [6] is an EC dialect that exhibits an absolute improvement of performance on computing the effects of events with respect to the original EC [15]. CEC does not operate on a working 'window' (WM), that is, it does not 'forget' any SDE. Although such a design decision guarantees that no information will be lost, it affects considerably the efficiency of CEC. As time progresses and SDE arrive at the system, the efficiency of CEC decreases. Consequently, in its current form, CEC cannot be used for run-time CE recognition as, at some point, the CE recognition times will fail to meet the user requirements.

If RTEC operated on the complete SDE history, as CEC and all other EC dialects do, in contrast to operating on WM, then the complexity of computing fluent intervals would increase substantially over time—replace m_{WM} in formulas (19) and (21) with $m(0, Q_i)$. The cost of the 'forget' mechanism—see formula (18)—is substantially smaller than the cost of computing fluent intervals taking into consideration the complete SDE history.

When an interval of a fluent is retracted, or asserted, as a result of the occurrence of a SDE that arrived in a non-chronological order, CEC propagates the update to the fluents whose validity may rely on such an interval. The reasoning performed by the modules propagating fluent assertions and retractions can be very costly, especially in real-world applications such as CTM, where there are many fluents that depend on many other fluents, and there are several rules defining fluents. Also, unlike RTEC, CEC does not support SDE revision. If CEC were to support this functionality, then the number of invocations of the modules propagating assertions and retractions would increase. Other approaches that follow this type of reasoning are, for example, [2, 3].

RTEC does not perform costly checks every time a fluent interval is asserted/retracted (due to the delayed arrival, or revision, of SDE). Instead, RTEC discards, at each query time Q_i, all fluent intervals in $(Q_i - WM, Q_i]$ and computes from scratch all intervals given the SDE that are available at Q_i and took place in $(Q_i - WM, Q_i]$.

Other EC dialects have been proposed in the literature. The Reactive Event Calculus [5], for example, is based on CEC but has not been evaluated yet, theoretically or experimentally. A well-known EC dialect is the Interval-based Event Calculus (IEC) [21]. In IEC it is not possible to recognise an 'on-going' CE, that is, a CE that started taking place at an earlier time and still holds. Moreover, although there seems to be in IEC some form of storing of sub-computations (concerning only time-points as fluent intervals are not represented) and event intervals, the possibilities of SDE arriving in a non-chronological order, and SDE revision, are not considered. Thus, in IEC it is not possible, for example, to update, that is, (partly) retract, the intervals of recognised CE due to SDE arriving with a delay or SDE revision.

Note that the assumptions of sorted input and no SDE revision are not restricted to IEC. Several event processing systems, such as [13, 10, 7, 9, 17], operate only under the assumption that SDE are temporally sorted. Such systems rely on components or network protocols that order SDE prior to feeding them to the CE recognition system. RTEC does not rely on such components/network protocols and may dynamically update the intervals of recognised CE. The applications mentioned in [18, 6, 2], as well as CTM in the Helsinki infrastructure, are but a few examples in which the SDE streams given to the CE recognition system cannot be assumed to be ordered, and/or may be revised.

7. SUMMARY AND FURTHER WORK

We presented RTEC, an EC dialect with novel implementation and 'windowing' techniques that allow for efficient CE recognition, scalable to large numbers of SDE and CE. RTEC may operate in the absence of SDE filtering modules, as it is only slightly affected by SDE that are irrelevant to the CE we want to recognise. Furthermore, RTEC remains efficient and scalable in applications where SDE arrive with a (variable) delay from, and are revised by, the underlying SDE detection system. RTEC may update the intervals of already recognised CE, and recognise new CE, due to SDE arriving with a delay or SDE revision.

RTEC has a formal semantics in terms of logic programming, while the formalisation of CE definitions, including the corresponding background knowledge (if any), is declar-

ative. Moreover, the interval manipulation constructs of RTEC, usable along side the standard EC rules, simplify CE definitions, and improve reasoning efficiency.

The complex CE definitions in the CTM application enabled us to perform a realistic experimental evaluation of RTEC. The evaluation showed that RTEC supports real-time reasoning in most of today's applications.

The are several directions for further work. First, we aim to extend RTEC by allowing for CE recognition under different sets of SDE. In some cases, for example, it may be required that different CE have different working memory sizes. For some CE it may be acceptable to sacrifice efficiency by having a larger working memory in order to minimise the possibility of losing information by discarding late SDE (revision). For other CE, efficiency may be more important and therefore the recognition of these CE may be based on a smaller working memory. Second, we aim to prove a set of properties satisfied by RTEC (see [5], for example) as well as demonstrate the verification of a CE definition library formalised in RTEC. Third, we aim to develop customisable consumption policies [7, 3] in order to use RTEC in application domains requiring event consumption. Such policies were not necessary in the case study presented here.

Acknowledgments

This work has been partially funded by EU, in the context of the PRONTO project (FP7-ICT 231738).

8. REFERENCES

[1] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13:177–203, 2004.

[2] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Retractable complex event processing and stream reasoning. In *RuleML Europe*, pages 122–137, 2011.

[3] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Real-time complex event recognition and reasoning — a logic programming approach. *Applied Artificial Intelligence*, 26(1–2):6–57, 2012.

[4] P. Bizzaro. Results of the survey on event processing use cases. Event Processing Technical Society, March 2011. `http://www.slideshare.net/pedrobizarro/epts-survey-results`.

[5] F. Chesani, P. Mello, M. Montali, and P. Torroni. A logic-based, reactive calculus of events. *Fundamenta Informaticae*, 105(1-2):135–161, 2010.

[6] L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12(3):359–382, 1996.

[7] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proceedings of Conference on Distributed-Event Based Systems (DEBS)*, pages 50–61, 2010.

[8] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 2011.

[9] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, pages 243–254, 2011.

[10] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. Candan. Runtime semantic query optimization for event stream processing. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 676–685, 2008.

[11] C. Dousson and P. L. Maigat. Chronicle recognition improvement using temporal focusing and hierarchisation. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 324–329, 2007.

[12] M. Eckert and F. Bry. Rule-based composite event queries: the language xchange^{eq} and its semantics. *Knowledge Information Systems*, 25(3):551–573, 2010.

[13] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex event processing over streams. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, 2007.

[14] A. Kimmig, B. Demoen, L. D. Raedt, V. S. Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.

[15] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–96, 1986.

[16] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1):1–49, 2009.

[17] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *Proceedings of international Conference on Distributed Event-Based Systems (DEBS)*, pages 291–302, New York, NY, USA, 2011. ACM.

[18] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 784–795, 2009.

[19] D. Luckham and R. Schulte. Event processing glossary — version 1.1. Event Processing Technical Society, July 2008. `http://www.ep-ts.com/`.

[20] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of SIGMOD Conference*, pages 193–206, 2009.

[21] A. Paschke and M. Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.

[22] A. Paschke and A. Kozlenkov. Rule-based event processing and reaction rules. In *Proceedings of RuleML*, volume LNCS 5858, pages 53–66. Springer, 2009.

[23] O. Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009.

[24] V. Shet, J. Neumann, V. Ramesh, and L. Davis. Bilattice-based logical reasoning for human detection. In *Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE, 2007.

[25] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.