An Event Calculus for Event Recognition

Alexander Artikis, Marek Sergot, and Georgios Paliouras

Abstract—Systems for symbolic event recognition accept as input a stream of time-stamped events from sensors and other computational devices, and seek to identify high-level composite events, collections of events that satisfy some pattern. RTEC is an Event Calculus dialect with novel implementation and 'windowing' techniques that allow for efficient event recognition, scalable to large data streams. RTEC supports the expression of rather complex events, such as 'two people are fighting', using simple primitives. It can operate in the absence of filtering modules, as it is only slightly affected by data that are irrelevant to the events we want to recognise. Furthermore, RTEC can deal with applications where event data arrive with a (variable) delay from, and are revised by, the underlying sources. RTEC can update already recognised events and recognise new events when data arrive with a delay or following data revision. We evaluate RTEC both theoretically, presenting a complexity analysis, and experimentally, using two real-world applications. The evaluation shows that RTEC can support real-time event recognition and is capable of meeting the performance requirements identified in a survey of event processing use cases.

Index Terms—event pattern matching, event processing, action language

1 Introduction

Systems for symbolic event recognition ('event pattern matching') accept as input a stream of time-stamped simple, derived events (SDE)s. A SDE ('low-level event') is the result of applying a computational derivation process to some other event, such as an event coming from a sensor [23]. Using SDEs as input, event recognition systems identify composite events (CE)s of interest—collections of events that satisfy some pattern. The 'definition' of a CE ('high-level event') imposes temporal and, possibly, atemporal constraints on its subevents, i.e. SDEs or other CEs. Consider e.g. the recognition of attacks on computer network nodes given the TCP/IP messages.

Numerous recognition systems have been proposed in the literature [12]. Recognition systems with a logic-based representation of CE definitions, in particular, have recently been attracting attention [5]. They exhibit a formal, declarative semantics, in contrast to other types of recognition system that usually rely on an informal and/or procedural semantics. Cugola and Margara [11] point out that almost all 'complex event processing languages', including [1], and several 'data stream processing languages', such as ESL [6] which extends CQL [3], lack a rigorous, formal semantics. Eckert and Bry [16] note that the semantics of 'event query languages' often are somewhat ad hoc, unintuitive and generally have an algebraic and less declarative flavour. Paschke and Kozlenkov [29] state that commercial 'production rule languages' lack a declarative semantics.

- *A. Artikis and G. Paliouras are with the NCSR Demokritos, Athens, Greece. E-mail: {a.artikis,paliourg}@iit.demokritos.gr*
- *M. Sergot is with the Department of Computing, Imperial College London, UK. E-mail: m.sergot@imperial.ac.uk*

Non-logic-based CE recognition systems have proven to be, overall, more efficient than logic-based ones. To address this issue, we present an efficient dialect of the Event Calculus [19], called 'Event Calculus for Run-Time reasoning' (RTEC). The Event Calculus is a logic programming formalism for representing and reasoning about events and their effects. RTEC includes novel implementation techniques for efficient CE recognition, scalable to large SDE and CE volumes. A form of caching stores the results of sub-computations in the computer memory to avoid unnecessary recomputations. A set of interval manipulation constructs simplify CE definitions and improve reasoning efficiency. A simple indexing mechanism makes RTEC robust to SDEs that are irrelevant to the CEs we want to recognise and so RTEC can operate without SDE filtering modules. Finally, a 'windowing' mechanism supports real-time CE recognition. One main motivation for RTEC is that it should remain efficient and scalable in applications where SDEs arrive with a (variable) delay from, or are revised by, the underlying SDE detection system: RTEC can update the already recognised CEs, and recognise new CEs, when SDEs arrive with a delay or following revision. The code of RTEC is available at ⟨http://users.iit.demokritos.gr/~a.artikis/EC.html⟩.

We evaluate RTEC theoretically, presenting a complexity analysis, and experimentally, using two real-world applications: city transport management (CTM) and public space surveillance (PSS) from video content. In CTM, public transport vehicles, such as buses and trams, are equipped with sensors that report on position, in-vehicle temperature, noise level and acceleration. Fixed sensors are mounted on intersections to report on traffic flow and density. Given such SDEs, the task is to inform the decision-making of transport officials by recognising CEs related to

traffic congestion, the punctuality of a vehicle, passenger and driver comfort, passenger and driver safety, and passenger satisfaction. In PSS, the SDEs are the 'short-term activities' detected on video frames—e.g. a person walking, running or being inactive. The aim then is to recognise 'long-term activities', i.e. short-term activity combinations, such as when a person leaves an object unattended, when two people are moving together, when they are having a meeting or fighting. The CE definitions for these applications are quite complex—a major benefit of RTEC is that it supports the expression of rather complex definitions—allowing for a realistic evaluation of the efficiency of RTEC. This is in contrast to the majority of related approaches where rather simple CE definitions are used for empirical analysis. Our evaluation shows that RTEC supports real-time CE recognition and is capable of meeting the performance requirements of most of today's applications as estimated by a recent survey of event processing use cases [7].

Organisation. Sections 2–4 present, respectively, RTEC, its reasoning algorithms and the complexity analysis. The experimental evaluation is given in Section 5. In Section 6 we put the work in context, while in Section 7 we summarise the presented work and outline directions for further research.

2 Event Calculus

Our system for CE recognition is based on an Event Calculus dialect. The Event Calculus [19] is a logic programming formalism for representing and reasoning about events and their effects. For the dialect introduced here, called RTEC, the time model is linear and includes integer time-points. Variables start with an upper-case letter, while predicates and constants start with a lower-case letter. 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. holdsAt$(F=V,T)$ represents that fluent F has value V at a particular time-point T. holdsFor$(F=V,I)$ represents that I is the list of the maximal intervals for which $F=V$ holds continuously. holdsAt and holdsFor are defined in such a way that, for any fluent F, holdsAt$(F=V,T)$ if and only if T belongs to one of the maximal intervals of I for which holdsFor$(F=V,I)$.

The happensAt predicate represents an instance of an event type. E.g. in public space surveillance happensAt$(appear(id_1),5)$ represents the occurrence of event type $appear(id_1)$ at time-point 5. When it is clear from context, we do not distinguish between an event and its type. An *event description* in RTEC includes rules 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 holdsAt and holdsFor predicates, as well as other, possibly atemporal, constraints. Table 1 summarises the RTEC predicates available to the event description developer. The last three items in the table are interval manipulation predicates specific to RTEC.

TABLE 1: Main predicates of RTEC.

Predicate	Meaning
happensAt(E,T)	Event E occurs at time T
holdsAt$(F=V,T)$	The value of fluent F is V at time T
holdsFor$(F=V,I)$	I is the list of the 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
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
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

We represent instantaneous SDEs and CEs by means of happensAt, while durative SDEs and CEs are represented as fluents. The majority of CEs are durative and, therefore, in CE recognition the task generally is to compute the maximal intervals for which a fluent representing a CE has a particular value continuously.

2.1 Simple Fluents

Fluents in RTEC are of two kinds: *simple* and *statically determined*. We assume, without loss of generality, that these types are disjoint. For a simple fluent F, $F=V$ holds at a particular time-point T if $F=V$ has been *initiated* by an event that has occurred at some time-point earlier than T, and has not been *terminated* at some other time-point in the meantime. This is an implementation of the law of inertia. To compute the *intervals* I for which $F=V$, i.e. 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 'broken'. The time-points at which $F=V$ is initiated are computed by means of domain-specific initiatedAt rules. The time-points at which $F=V$ is 'broken' are computed as follows:

$$\text{broken}(F=V,\ T_s,\ T) \leftarrow \text{terminatedAt}(F=V,\ T_f),\quad T_s < T_f \leq T \tag{1}$$

$$\text{broken}(F=V_1,\ T_s,\ T) \leftarrow \text{initiatedAt}(F=V_2,\ T_f),\quad T_s < T_f \leq T,\quad V_1 \neq V_2 \tag{2}$$

broken$(F=V,T_s,T)$ represents that a maximal interval starting at T_s for which $F=V$ holds continuously is terminated at some time T_f such that $T_s<T_f\leq T$. Similar to initiatedAt, terminatedAt rules are domain-specific (examples are presented below). According to rule (2), if $F=V_2$ is initiated at T_f then effectively $F=V_1$ is

terminated at time T_f, for all other possible values V_1 of F. Rule (2) ensures therefore that a fluent cannot have more than one value at any time. We do not insist that a fluent must have a value at every time-point. There is a difference between initiating a Boolean fluent $F = \text{false}$ and terminating $F = \text{true}$: the former implies, but is not implied by, the latter.

In city transport management, officials are interested in identifying tendencies towards traffic congestion. Consider the following formalisation:

$$\begin{aligned}&\text{initiatedAt}(\mathit{densityTrend}(S) = \mathit{increasing},\ T) \leftarrow\\ &\quad \text{happensAt}(\mathit{traffic}(S, \mathit{Flow}, \mathit{Density}),\ T),\\ &\quad \text{happensAt}(\mathit{traffic}(S, \mathit{Flow}', \mathit{Density}'),\ T{+}60),\\ &\quad \mathit{Density}' > \mathit{Density}{+}\mathit{Density}{\times}0.2\end{aligned} \tag{3}$$

$\mathit{traffic}(S, \mathit{Flow}, \mathit{Density})$ is an instantaneous SDE reporting traffic flow and density in the junction where sensor S is mounted. Each such sensor reports on flow and density every 60 seconds. According to rule (3), traffic density is said to be increasing if in two consecutive SDEs there is a rise of more than 20% in the density value. The maximal intervals during which $\mathit{densityTrend}(S) = \mathit{increasing}$ holds continuously are computed using the built-in RTEC predicate holdsFor from rule (3) and other similar rules, not shown here, defining the remaining values of $\mathit{densityTrend}$. When density is increasing (resp. traffic flow is decreasing) transport officials usually take proactive measures against traffic congestion.

$\text{initiatedAt}(F = V, T)$ does not necessarily imply that $F \neq V$ at T. Similarly, $\text{terminatedAt}(F = V, T)$ does not necessarily imply that $F = V$ at T. Suppose that $F = V$ is initiated at time-points 10 and 20 and terminated at time-points 25 and 30 (and at no other time-points). In that case $F = V$ holds at all T such that $10 < T \leq 25$.

In addition to constraints on events, initiatedAt and terminatedAt predicates in the bodies (antecedents) of rules may specify constraints on fluents. Consider the following example from public space surveillance:

$$\begin{aligned}&\text{initiatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{start}(\mathit{walking}(P_1) = \text{true}),\ T),\\ &\quad \text{holdsAt}(\mathit{walking}(P_2) = \text{true},\ T),\\ &\quad \text{holdsAt}(\mathit{close}(P_1, P_2) = \text{true},\ T)\\ &\text{initiatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{start}(\mathit{walking}(P_2) = \text{true}),\ T),\\ &\quad \text{holdsAt}(\mathit{walking}(P_1) = \text{true},\ T),\\ &\quad \text{holdsAt}(\mathit{close}(P_1, P_2) = \text{true},\ T)\\ &\text{initiatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{start}(\mathit{close}(P_1, P_2) = \text{true}),\ T),\\ &\quad \text{holdsAt}(\mathit{walking}(P_1) = \text{true},\ T),\\ &\quad \text{holdsAt}(\mathit{walking}(P_2) = \text{true},\ T)\\ &\text{terminatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{end}(\mathit{walking}(P_1) = \text{true}),\ T)\\ &\text{terminatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{end}(\mathit{walking}(P_2) = \text{true}),\ T)\\ &\text{terminatedAt}(\mathit{moving}(P_1, P_2) = \text{true},\ T) \leftarrow\\ &\quad \text{happensAt}(\text{end}(\mathit{close}(P_1, P_2) = \text{true}),\ T)\end{aligned} \tag{4}$$

$\mathit{walking}$ is a durative SDE detected on video frames. $\text{start}(F = V)$ (resp. $\text{end}(F = V)$) is a built-in RTEC event taking place at each starting (ending) point of each maximal interval for which $F = V$ holds continuously. $\mathit{close}(A, B) = \text{true}$ when the distance between A and B does not exceed some threshold of pixel positions. The above formalisation states that P_1 is moving with P_2 when they are walking close to each other.

One of the main attractions of RTEC is that it makes available the power of logic programming to express complex temporal and atemporal constraints, as conditions in initiatedAt and terminatedAt rules for durative CEs, and happensAt rules for instantaneous CEs. E.g. standard event algebra operators, such as sequence, disjunction, parallelism, etc, may be expressed in a RTEC event description.

2.2 Statically Determined Fluents

In addition to the domain-independent definition of holdsFor, an event description may include domain-specific holdsFor rules, used to define the values of a fluent F in terms of the values of other fluents. We call such a fluent F *statically determined*. holdsFor rules of this kind make use of interval manipulation constructs—see the last three items of Table 1. Consider, e.g. moving as in rules (4) but defined instead as a statically determined fluent:

$$\begin{aligned}&\text{holdsFor}(\mathit{moving}(P_1, P_2) = \text{true},\ I) \leftarrow\\ &\quad \text{holdsFor}(\mathit{walking}(P_1) = \text{true},\ I_1),\\ &\quad \text{holdsFor}(\mathit{walking}(P_2) = \text{true},\ I_2),\\ &\quad \text{holdsFor}(\mathit{close}(P_1, P_2) = \text{true},\ I_3),\\ &\quad \text{intersect\_all}([I_1, I_2, I_3],\ I)\end{aligned} \tag{5}$$

According to the above rule, the list I of maximal intervals during which P_1 is moving with P_2 is computed by determining the list I_1 of maximal intervals during which P_1 is walking, the list I_2 of maximal intervals during which P_2 is walking, the list I_3 of maximal intervals during which P_1 is close to P_2, and then calculating the list I representing the intersections of the maximal intervals in I_1, I_2 and I_3.

RTEC provides three interval manipulation constructs: union_all, intersect_all and relative_complement_all. $\text{union\_all}(L, I)$ computes the list I of maximal intervals representing the union of maximal intervals of the lists of list L. For instance:

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

A term of the form (T_s, T_e) in RTEC represents the closed-open interval $[T_s, T_e)$. I in $\text{union\_all}(L, I)$ is a list of maximal intervals that includes each time-point that is part of at least one list of L. The implementation of all interval manipulation constructs is available with the code of RTEC.

$\text{intersect\_all}(L, I)$ computes the list I of maximal intervals such that I represents the intersection of maximal intervals of the lists of list L, as, e.g.:

$$\text{intersect\_all}([[(26, 31)], [(21, 26), (30, 40)]],\ [(30, 31)])$$

I in intersect_all(L, I) is a list of maximal intervals that includes each time-point that is part of all lists of L.

relative_complement_all(I', L, I) computes the list I of maximal intervals such that I represents the relative complements of the list of maximal intervals I' with respect to the maximal intervals of the lists of list L. Below is an example of relative_complement_all:

relative_complement_all$([(5,20),(26,50)],$
$[[(1,4),(18,22)],[(28,35)]],\ [(5,18),(26,28),(35,50)])$

I in relative_complement_all(I', L, I) is a list of maximal intervals that includes each time-point of I' that is not part of any list of L. The CE definition below uses all interval manipulation constructs of RTEC:

$$\begin{aligned}
&\text{holdsFor}(\mathit{fighting}(P_1,P_2)=\text{true},\ I) \leftarrow \\
&\quad \text{holdsFor}(\mathit{abrupt}(P_1)=\text{true},\ I_1), \\
&\quad \text{holdsFor}(\mathit{abrupt}(P_2)=\text{true},\ I_2), \\
&\quad \text{holdsFor}(\mathit{close}(P_1,P_2)=\text{true},\ I_3), \\
&\quad \text{union_all}([I_1,I_2],\ I_4),\ \ \text{intersect_all}([I_4,I_3],\ I_5), \\
&\quad \text{holdsFor}(\mathit{inactive}(P_1)=\text{true},\ I_6), \\
&\quad \text{holdsFor}(\mathit{inactive}(P_2)=\text{true},\ I_7), \\
&\quad \text{relative_complement_all}(I_5,\ [I_6,I_7],\ I)
\end{aligned} \tag{6}$$

In the public space surveillance application, *abrupt* and *inactive* are durative SDEs detected on video frames. According to rule (6), two people are assumed to be *fighting* as long as at least one of them is moving abruptly, the other is not inactive, and they are close.

The interval manipulation constructs of RTEC support the following type of definition: for all time-points T, $F=V$ holds at T if and only if some Boolean combination of fluent-value pairs holds at T. For a wide range of fluents, this is a much more concise definition than the traditional style of Event Calculus representation, i.e. identifying the various conditions under which the fluent is initiated and terminated so that maximal intervals can then be computed using the domain-independent holdsFor. Compare, e.g. the statically determined and simple fluent representations of *moving* in rules (5) and (4) respectively.

The interval manipulation constructs of RTEC can also lead to much more efficient computation. We will return to that point in Section 4.

2.3 Semantics

CE definitions are (locally) stratified logic programs [30]. We restrict attention to *hierarchical* definitions, those where it is possible to define a function *level* that maps all fluent-values $F=V$ and all events to the non-negative integers as follows. Events and statically determined fluent-values $F=V$ of level 0 are those whose happensAt and holdsFor definitions do not depend on any other events or fluents. In CE recognition, they represent the input SDEs. There are no fluent-values $F=V$ of simple fluents F in level 0. Events and simple fluent-values of level n are defined in terms of at least one event or fluent-value of level $n-1$ and a possibly empty set of events and fluent-values from levels lower than $n-1$. Statically determined fluent-values of level n are defined in terms of at least one fluent-value of level $n-1$ and a possibly empty set of fluent-values from levels lower than $n-1$. Note that fluent-values $F=V_i$ and $F=V_j$ for $V_i \neq V_j$ could be mapped to different levels. For simplicity however, and without loss of generality, a fluent F itself is either simple or statically determined but not both. The CE definitions of city transport management and public space surveillance, i.e. the holdsFor definitions of statically determined fluents, initiatedAt and terminatedAt definitions of simple fluents and happensAt definitions of events, are available with the RTEC code.

3 Run-Time Recognition

CE recognition has to be efficient enough to support real-time decision-making, and scale to very large numbers of SDEs and CEs. SDEs may not necessarily arrive at the CE recognition system in a timely manner, i.e. there may be a (variable) delay between the time at which SDEs take place and the time at which they arrive at the CE recognition system. Moreover, SDEs may be revised, or even completely discarded in the future, as in the case where the parameters of a SDE were originally computed erroneously and are subsequently revised, or in the case of 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.

RTEC performs CE recognition by 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 Q_i the SDEs that fall within a specified interval—the 'working memory' (*WM*) or 'window'—are taken into consideration. All SDEs that took place before or at $Q_i - WM$ are discarded. This is to make the cost of CE recognition dependent only on the *WM* size and not on the complete SDE history. The *WM* size, and the temporal distance between two consecutive query times — the 'step' ($Q_i - Q_{i-1}$) — are set by the user.

At Q_i, the maximal intervals computed by RTEC are those that can be derived from SDEs that occurred in the interval $(Q_i - WM, Q_i]$, as recorded at time Q_i. When *WM* is longer than the inter-query step, i.e., when $Q_i - WM < Q_{i-1} < Q_i$, it is possible that an SDE occurs in the interval $(Q_i - WM, Q_{i-1}]$ but arrives at RTEC only after Q_{i-1}; its effects are taken into account at query time Q_i. And similarly for SDEs that took place in $(Q_i - WM, Q_{i-1}]$ and were subsequently revised after Q_{i-1}. In the common case that SDEs arrive at RTEC with delays, or there is SDE revision, it is preferable therefore to make *WM* longer than the inter-query step. Note that information may still be lost. Any SDEs arriving or revised between Q_{i-1} and Q_i are discarded at Q_i if they took place before or at $Q_i - WM$. To reduce the possibility of losing

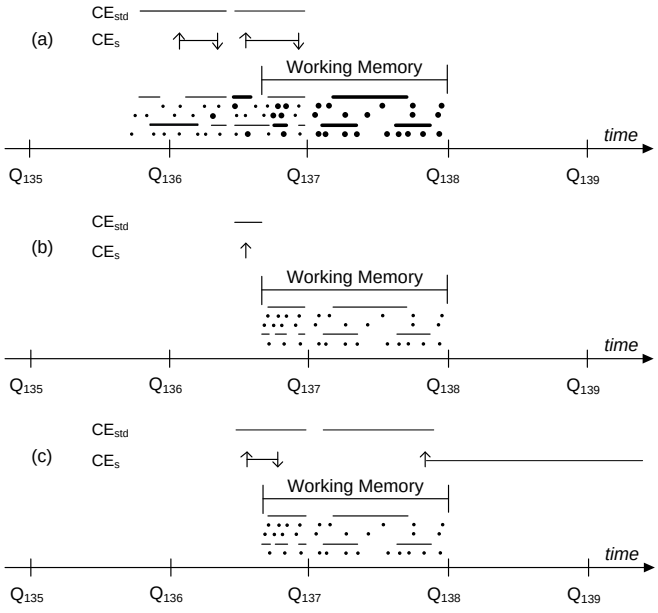


Fig. 1: Windowing in RTEC.

information, one may increase the *WM* size. Doing so, however, decreases recognition efficiency. In what follows we give an example and a detailed account of the 'windowing' algorithm of RTEC.

3.1 Illustrative Example

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

- CE_s, represented as a simple fluent (see Section 2.1). The starting and ending points, and the maximal intervals of CE_s are displayed above *WM* in Figure 1.
- CE_{std}, represented as a statically determined fluent (see Section 2.2). For the example, the maximal intervals of CE_{std} are defined to be the union of the maximal intervals of the two durative SDEs in Figure 1. The maximal intervals of CE_{std} are displayed above the CE_s intervals.

For simplicity, we assume that both CE_s and CE_{std} are defined only in terms of SDE, i.e. they are not defined in terms of other CEs.

Figure 1 shows the steps that are followed in order to recognise CEs at an arbitrary query time, say Q_{138}. Figure 1(a) shows the state of RTEC as computation begins at Q_{138}. All SDEs that took place before or at $Q_{137} - WM$ were retracted at Q_{137}. The thick lines and dots represent the SDEs that arrived at RTEC between Q_{137} and Q_{138}; some of them took place before Q_{137}. Figure 1(a) also shows the maximal intervals for the CE fluents CE_s and CE_{std} that were computed and stored at Q_{137}.

The CE recognition process at Q_{138} considers the SDEs that took place in $(Q_{138} - WM, Q_{138}]$. All SDEs that took place before or at $Q_{138} - WM$ are discarded, as shown in Figure 1(b). For durative SDEs that started before $Q_{138} - WM$ and ended after that time, RTEC retracts the sub-interval up to and including $Q_{138} - WM$. Figure 1(b) shows the interval of a SDE that is partially retracted in this way.

Now consider CE intervals. At Q_i some of the maximal intervals computed at Q_{i-1} might have become invalid. This is because some SDEs occurring in $(Q_i - WM, Q_{i-1}]$ might have arrived or been revised after Q_{i-1}: their existence could not have been known at Q_{i-1}. Determining which CE intervals should be (partly) retracted in these circumstances can be computationally very expensive. See Section 6 for a discussion. We find it simpler, and more efficient, to discard all CE intervals in $(Q_i - WM, Q_i]$ and compute all intervals from scratch in that period. CE intervals that have ended before or at $Q_i - WM$ are discarded. Depending on the user requirements, these intervals may be stored in a database for retrospective inspection of the activities of a system.

In Figure 1(b), the earlier of the two maximal intervals computed for CE_{std} at Q_{137} is discarded at Q_{138} since its endpoint is before $Q_{138} - WM$. The later of the two intervals overlaps $Q_{138} - WM$ (an interval 'overlaps' a time-point t if the interval starts before or at t and ends after or at that time) and is partly retracted at Q_{138}. Its starting point could not have been affected by SDEs arriving between $Q_{138} - WM$ and Q_{138} but its endpoint has to be recalculated. Accordingly, the sub-interval from $Q_{138} - WM$ is retracted at Q_{138}.

In this example, the maximal intervals of CE_{std} are determined by computing the union of the maximal intervals of the two durative SDEs shown in Figure 1. At Q_{138}, only the SDE intervals in $(Q_{138} - WM, Q_{138}]$ are considered. In the example, there are two maximal intervals for CE_{std} in this period as can be seen in Figure 1(c). The earlier of them has its startpoint at $Q_{138} - WM$. Since that abuts the existing, partially retracted sub-interval for CE_{std} whose endpoint is $Q_{138} - WM$, those two intervals are amalgamated into one continuous maximal interval as shown in Figure 1(c). In this way, the endpoint of the CE_{std} interval that overlapped $Q_{138} - WM$ at Q_{137} is recomputed to take account of SDEs available at Q_{138}. (In this particular example, it happens that the endpoint of this interval is the same as that computed at Q_{137}. That is merely a feature of this particular example. Had CE_{std} been defined e.g. as the *intersection* of the maximal intervals of the two durative SDE, then the intervals of CE_{std} would have changed in $(Q_{138} - WM, Q_{137}]$.)

Figure 1 also shows how the intervals of the simple fluent CE_s are computed at Q_{138}. Arrows facing upwards (downwards) denote the starting (ending) points of CE_s intervals. First, in analogy with the treatment of statically determined fluents, the ear-

lier of the two CE_s intervals in Figure 1(a), and its start and endpoints, are retracted. They occur before $Q_{138}-WM$. The later of the two intervals overlaps $Q_{138}-WM$. The interval is retracted, and only its starting point is kept; its new endpoint, if any, will be recomputed at Q_{138}. See Figure 1(b). For simple fluents, it is simpler, and more efficient, to retract such intervals completely and reconstruct them later from their start and endpoints by means of 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 statically determined fluents.

The second step for CE_s at Q_{138} is to calculate its starting and ending points by evaluating the relevant initiatedAt and terminatedAt rules. For this, we only consider SDEs that took place in $(Q_{138}-WM, Q_{138}]$. Figure 1(c) shows the starting and ending points of CE_s in $(Q_{138}-WM, Q_{138}]$. The last ending point of CE_s that was computed at Q_{137} was invalidated in the light of the new SDEs that became available at Q_{138} (compare Figures 1(c)–(a)). Moreover, another ending point was computed at an earlier time.

Finally, in order to recognise CE_s at Q_{138} we use the domain-independent holdsFor to calculate the maximal intervals of CE_s given its starting and ending points. The later of the two CE_s intervals computed at Q_{137} became shorter when re-computed at Q_{138}. The second interval of CE_s at Q_{138} is open: given the SDEs available at Q_{138}, we say that CE_s holds *since* time t, where t is the last starting point of CE_s.

The discussion above showed that, when SDEs arrive with a variable delay, CE intervals computed at an earlier query time may be (partly) retracted at the current or a future query time. (And similarly if SDEs are revised.) Depending on the application requirements, RTEC may be set to report:

- CEs as soon as they are recognised, even if their intervals may be (partly) retracted in the future.
- CEs whose intervals may be partly, but not completely, retracted in the future, i.e. CEs whose intervals overlap $Q_{i+1}-WM$.
- CEs whose intervals will not be even partly retracted in the future, i.e. CEs whose intervals end before or at $Q_{i+1}-WM$.

The example used for illustration shows how RTEC performs CE recognition. To support real-time reasoning, at each query time Q_i all SDEs that took place before or at Q_i-WM are discarded. To handle efficiently delayed SDEs and SDE revision, CE intervals within WM are computed from scratch. At Q_i, the computed maximal CE intervals are those that can be derived from SDEs that occurred in the interval $(Q_i-WM, Q_i]$, as recorded at time Q_i. For completeness, RTEC amalgamates the computed intervals to any intervals ending at Q_i-WM. In the section below we present the CE recognition algorithm, and in Section 4 we discuss the complexity of RTEC.

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

```
1: retract(sdFList(Index, CE_std, OldI, OldPE))
2: amalgamate(OldPE, OldI, OldList)
3: if Start, End : [Start, End) ∈ OldList ∧
       End > Q_i − WM ∧ Start ≤ Q_i − WM then
4:         PE: =[(Start, Q_i − WM + 1)]
5:     else PE: =[ ]
6: end if
7: holdsForSDFluent(CE_std, I)
8: assert(sdFList(Index, CE_std, I, PE))
```

3.2 RTEC Operation

After 'forgetting' SDEs, i.e. after retracting SDE intervals taking place before or at Q_i-WM, RTEC computes and stores the intervals of each CE of interest. At the end of CE recognition at each query time Q_i, all computed fluent intervals are stored in the computer memory as simpleFList and sdFList assertions. I in sdFList($Index, CE_{std}, I, PE$) (resp. simpleFList($Index, CE_s, I, PE$)) represents the intervals of statically determined fluent CE_{std} (simple fluent CE_s) starting in $(Q_i-WM, Q_i]$, sorted in temporal order. PE stores the interval, if any, ending at Q_i-WM. The first argument in sdFList (simpleFList) 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. When the user queries the maximal intervals of a fluent, 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 $[Q_i-WM, Q_i]$. Next, we present how RTEC computes and stores the maximal intervals of fluents at each Q_i. Computing and storing the time-points of events representing instantaneous CEs is simpler and omitted here to save space.

Listing 1 shows the pseudo-code of recogniseSDFluent, the procedure for computing and storing the intervals of statically determined fluents. First, RTEC retrieves from sdFList the maximal intervals of a statically determined fluent CE_{std} computed at Q_{i-1} and checks if there is such an interval that overlaps Q_i-WM (lines 1–6). In Listing 1, $OldI$ represents the intervals of CE_{std} 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 $OldList$ (line 2). If there is an interval $[Start, End)$ in $OldList$ that overlaps Q_i-WM, then the sub-interval $[Start, Q_i-WM+1)$ is retained. See PE in Listing 1. All intervals in $OldList$ after Q_i-WM are discarded.

At the second step of recogniseSDFluent, RTEC evaluates holdsForSDFluent rules to compute the CE_{std} intervals from SDEs recorded as occurring in $(Q_i-WM, Q_i]$ (line 7). Prior to the run-time recognition process, RTEC has transformed holdsFor rules concerning stati-

cally determined fluents into holdsForSDFluent rules, in order to avoid unnecessary holdsFor rule evaluations. The intervals of CE_{std} computed at the previous query time Q_{i-1} are not taken into consideration in the evaluation of holdsForSDFluent rules. The computed list of intervals I of CE_{std}, along with PE, are stored in sdFList (line 8), replacing the intervals computed at Q_{i-1}. (Recall that, when the user queries the maximal intervals of a fluent, RTEC amalgamates PE with the intervals in I.)

recogniseSimpleFluent, the procedure for computing and storing simple fluent intervals, also has two parts. First, RTEC checks if there is a maximal interval of the fluent CE_s that overlaps $Q_i{-}WM$. If there is such an interval then it will be discarded, while its starting point will be kept. Second, RTEC computes the starting points of CE_s, without considering the starting points calculated at Q_{i-1}. The starting points are given to holdsForSimpleFluent, into which holdsFor calls computing the maximal intervals of simple fluents are translated at compile time. This program is defined as follows:

$$\begin{aligned} &\mathsf{holdsForSimpleFluent}([\,],\ \_,\ [\,])\\ &\mathsf{holdsForSimpleFluent}(SP,\ CE_s,\ I) \leftarrow\\ &\quad SP \neq [\,],\ \ \mathsf{computeEndingPoints}(CE_s,\ EP),\\ &\quad \mathsf{makeIntervals}(SP,\ EP,\ I) \end{aligned} \tag{7}$$

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

4 Complexity Analysis

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 therefore $m_{WM}/2$. Table 2 summarises the notation employed in this section.

TABLE 2: Complexity Analysis Notation.

Notation	Meaning
$m(S,E)$	Number of time-points in the interval $(S,E]$
m_{WM}	Number of time-points in the working memory *WM*
n	Number of SDE types
f	Number of fluent types
e	Number of event types
k	Number of interval manipulation constructs
l	Number of rules defining a simple fluent

4.1 Forget Mechanism

At each query time Q_i, RTEC first 'forgets' all available SDEs ending before or at $Q_i{-}WM$. In the common case that SDEs arrive with a variable delay, RTEC goes through the complete list of SDEs available at Q_i. In the worst case, all SDEs 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

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

where n denotes the number of SDE types. This is the cost of going through the SDEs in $(0,Q_i]$ and retracting those in $(0,Q_i{-}WM]$. This situation may occur at most once since all SDEs 'forgotten' at Q_i are not available after Q_i. In practice, the cost of the 'forget' mechanism is bound by approximately

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

i.e. the SDEs that took place before or at $Q_{i-1}{-}WM$ are typically retracted at Q_{i-1} and are not available at Q_i.

4.2 Statically Determined Fluents

First, 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

$$\mathcal{O}\big(m_{WM}/2 + 1\big) \quad . \tag{8}$$

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

Second, RTEC evaluates a holdsForSDFluent rule. The cost of evaluating such a rule is bound 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 holdsForSDFluent represents a SDE or a CE. In either case, RTEC simply retrieves the fluent intervals from the computer memory. RTEC performs recognition bottom-up and thus the intervals of all CEs 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 argument of simpleFList (sdFList) is a list of 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 SDEs used in CE definitions. 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 union for calculating the union of two lists of maximal intervals. The cost of 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 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 union are disjoint. Assuming x lists of maximal intervals of size y, the cost of union_all is bound by:

$$\mathcal{O}(\overbrace{2y}^{1st\ \text{union}} + \overbrace{2y+y}^{2nd\ \text{union}} + \ldots + \overbrace{2y+y+\ldots+y}^{x-1th\ \text{union}}) = \mathcal{O}\left(y\left(\frac{x(x+1)}{2}-1\right)\right) \quad . \tag{9}$$

To compute the intersection of a list of lists of maximal intervals, RTEC recursively uses intersection for calculating the intersection of two lists of maximal intervals. Like union, the cost of intersection is limited by the sum of the sizes of the two lists, if each list is sorted. The size of the output list of intersection is bound by the size of the longest input list. The cost of intersect_all is bound by:

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

relative_complement_all(I', L, I) recursively uses relative_complement 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 relative_complement is limited by the sum of the sizes of the two input lists. Moreover, the size of the output list of relative_complement 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 (SDEs and CEs)—in the worst case this is the number of fluent types of the event description—and k interval manipulation constructs, the cost of evaluating such a rule is bound by

$$\mathcal{O}\left(f + k\,\frac{m_{WM}}{2}\left(\frac{f(f+1)}{2}-1\right)\right) \quad . \tag{10}$$

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 (9)). 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 $m_{WM}/2$.

4.3 Simple Fluents

The first step of recogniseSimpleFluent has the same cost as the first step of recogniseSDFluent—see formula (8). At the second step, RTEC computes the maximal intervals for which $F=V$ holds continuously. The cost of this step is limited by the sum of the cost of computing the starting points of $F=V$, the cost of computing the ending points of $F=V$, and the cost of makeIntervals (see formalisation (7)). Starting and ending points are computed by evaluating initiatedAt and terminatedAt rules. Assume that there are e events in the body of an initiatedAt/terminatedAt rule. e is bound by the number of event types of an event description. Evaluating a happensAt predicate expressing an event in the body of an initiatedAt/terminatedAt rule requires retrieving the event time-points from the computer memory.

Assume also that there are f fluents in the body of an initiatedAt/terminatedAt rule. In the worst case, this is the number of fluent types of the event description. Fluents are represented by means of holdsAt in the body of initiatedAt/terminatedAt. Evaluating holdsAt$(G=U, T)$ in the body of an initiatedAt/terminatedAt rule requires retrieving the intervals for which fluent G has value U that are stored in simpleFList or sdFList, and checking whether T belongs to these intervals. Each fluent has at most $m_{WM}/2$ maximal intervals stored in simpleFList/sdFList. The cost of computing the starting and ending points of $F=V$, therefore, is bound by

$$\mathcal{O}\big(m_{WM}\, l(e + f + f m_{WM}/2)\big) \quad , \tag{11}$$

where l is the number of initiatedAt/terminatedAt rules defining $F=V$. In the worst case, RTEC will evaluate all l rules defining $F=V$ m_{WM} times. For each time-point in $(Q_i{-}WM, Q_i]$, RTEC will check whether each event in the body of an initiatedAt/terminatedAt rule has taken place, and for every fluent in the body of such a rule, it will check whether that time-point belongs to one of the maximal intervals of that body fluent.

In practice, l, e and f are small—e.g. in rule (3), $f=0$. Moreover, an initiatedAt/terminatedAt rule is evaluated considerably fewer times than m_{WM} — such a rule is evaluated as many times as the number of instances of the first event in the body of the rule.

Finally, makeIntervals sorts the lists of starting and ending points, and then computes maximal intervals by retrieving, for every starting point T_s, the first ending point T_f, if any, after T_s. The sum of starting and ending points of a fluent is bound by m_{WM}. The cost of the first step of makeIntervals, therefore, is bound by $\mathcal{O}(m_{WM} \log m_{WM})$ while the cost of the second step is bound by $\mathcal{O}(m_{WM})$.

Given the above, the total cost of computing the maximal intervals of a simple fluent is bound by

$$\mathcal{O}\big(m_{WM}(l\,(e + f + f m_{WM}/2) + \log m_{WM} + 1)\big) \quad .$$

4.4 Simple vs Statically Determined Fluents

The presented complexity analysis may guide the decision to formalise a CE as a simple fluent vs a statically determined one. Consider e.g. the *moving* CE from public space surveillance, represented in rule (5) as a statically determined fluent. To compute the cost of this representation, we make the following substitutions in formula (10): $f=3$ (3 body fluents) and $k=1$ (1 interval manipulation construct). Therefore, the cost of this representation is bound by

$$\mathcal{O}(3 + 5m_{WM}/2) \quad .$$

The simple fluent representation of *moving*—see formalisation (4)—requires 6 initiatedAt and terminatedAt

rules with 3 fluents and no events in the body (recall that the start and end built-in events are defined in terms of fluents). From formula (11) we have that the cost of computing the starting and ending points of $moving$ is bound by

$$\mathcal{O}\big(6m_{WM}(3+3m_{WM}/2)\big) \quad .$$

To be precise, this cost is bound by

$$\mathcal{O}\big(3m_{WM}(3+3m_{WM}/2)\big)$$

because the initiatedAt and terminatedAt rules of $moving$ are specified in such a way that at most 3 of them may be evaluated at any time. A statically determined fluent representation of $moving$, therefore, leads to more efficient reasoning than a simple fluent representation. (For simplicity, we omit the cost of makeIntervals.) Similarly, we could have shown, for some other fluent, why a simple fluent representation leads to more efficient reasoning than a statically determined fluent representation.

5 Experimental Results

We present experimental results on city transport management (CTM) and public space surveillance (PSS). The experiments were performed on a computer with eight Intel i7 950@3.07GHz processors and 12GiB RAM, running Ubuntu Linux 12.04 and YAP Prolog 6.2.2. Each CE recognition time displayed in this section is the average of 30 runs, each produced by replaying the stream of SDEs from the recorded dataset. Each SDE is sent to RTEC according to its timestamp plus a randomly chosen non-negative delay, while RTEC computed continuous queries according to specified working memory and step sizes. For instance, when the working memory and step are both set to 10 seconds, RTEC performs CE recognition every 10 seconds using the SDEs with a timestamp in the last 10 seconds.

As explained in Section 3, at each query-time Q_i, RTEC first discards all CE intervals that overlap Q_i-WM, and then computes all such intervals from scratch. This is in order to avoid costly checks every time a fluent interval is asserted/retracted due to delayed SDE arrival or revision. For this reason RTEC does not benefit from a warm-up period. Computed SDE and CE intervals that overlap Q_i-WM are recorded in the computer memory. It is assumed that the computer memory is sufficient for this amount of information (as it is in all the experiments reported in this section). We have not developed a technique for dealing with the possibility of running out of memory. All the CE definitions and the datasets on which the experiments were performed are available with the RTEC code.

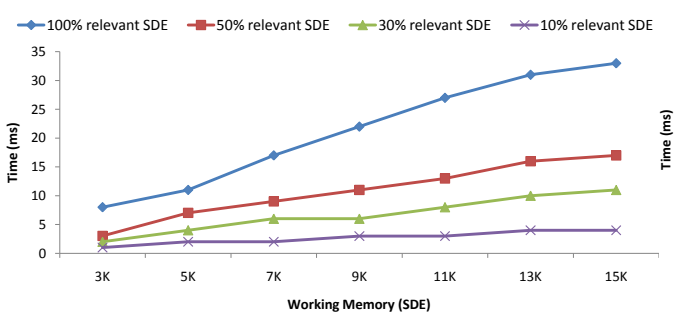
5.1 City Transport Management

To evaluate RTEC on CTM we used two datasets: the data of the PRONTO project [4], and the transport and traffic streams from Dublin, Ireland ⟨www.dublinked.ie⟩. In PRONTO, the task was to recognise CEs related to the punctuality of a public transport vehicle (bus or tram), driving style and quality, passenger and driver comfort, and passenger satisfaction. These CEs were requested by the public transport control centre of Helsinki, Finland, in order to support resource management. Buses and trams were equipped with sensors for detecting SDEs related to changes in position, acceleration, in-vehicle temperature, noise level and passenger density. At the time of the project, the available datasets included only a subset of the anticipated SDE types as some SDE detection components were not functional yet. In order to provide a more stringent evaluation, therefore, we performed experiments on synthetic data that are considerably more demanding than the real data [4]. Each synthetic stream contains instances of every SDE type (equal numbers of each). SDEs are not chronologically ordered—in CTM there are SDE delays.

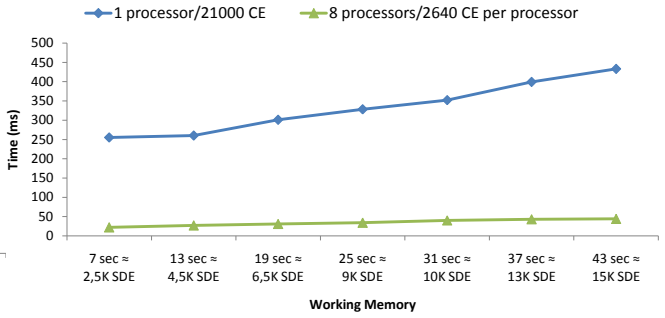
CE recognition for a single vehicle. Figure 2(a) shows experimental results regarding CE recognition for a single vehicle (bus or tram). These were intended to test the effects of varying the *WM* size and the tolerance of RTEC to irrelevant SDEs. The figure shows the results of four sets of experiments. In the first, only 10% of the SDEs concern the vehicle for which we perform CE recognition. In the second and the third, 30% and 50% respectively of the SDEs concern this vehicle. In the fourth case, all available SDEs concern it. In every case, RTEC computes and stores the intervals of 20 CE—fluent and event—types (each vehicle is associated with 20 CEs). We also varied the *WM* size. Figure 2(a) shows results of experiments in which *WM* varies from 3,000 to 15,000 SDEs. The times displayed in this figure show average CE recognition time in CPU milliseconds (ms).

RTEC employs a very simple indexing mechanism: 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(a), the presence of irrelevant SDEs 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 SDEs and we can distribute recognition tasks across multiple machines and processors (as demonstrated below).

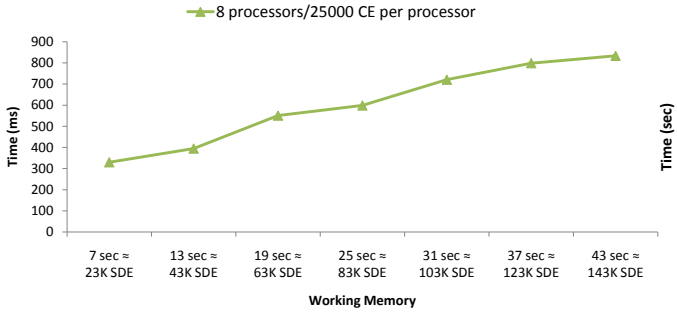
CE recognition for multiple vehicles. The next set of experiments concerns CE recognition at rush hour in Helsinki. At most 1,050 vehicles, i.e. 80% of the total number of available vehicles, operate at the same time in Helsinki during rush hour. It is estimated that no more than 1 SDE per 3 seconds (sec) can be detected on a single vehicle—no more than 350 SDEs can be detected per sec on the 1,050 operating vehicles. We were thus able to test RTEC under the maximum expected SDE frequency. Figure 2(b) presents the results of two sets of experiments. First, we used a single processor



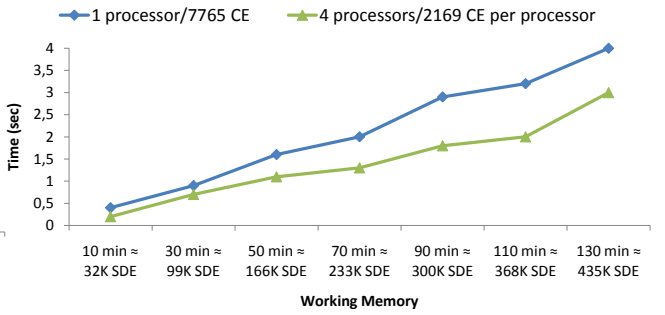
(a) PRONTO: CE recognition for a single vehicle.



(b) PRONTO: CE recognition for 1,050 vehicles.



(c) PRONTO: CE recognition for 10,000 vehicles.



(d) Dublin: CE recognition for 912 buses and 1,675 SCATS.

Fig. 2: Event Recognition for City Transport Management.

to perform CE recognition for all 1,050 vehicles. In this case, the intervals of 21,000 CEs (1,050 vehicles×20 CEs per vehicle) are computed and stored. Second, we used all eight processors of the computer in parallel. Each instance of RTEC running on a processor was set to perform CE recognition for 132 vehicles, and computed and stored the intervals of 2,640 CEs. We emphasize that the input data was the same in all sets of experiments: each processor receives SDEs coming from *all* 1,050 vehicles—i.e. there was no SDE filtering to restrict the input relevant for each processor. We rely only on the indexing mechanism to pick out relevant SDEs from the stream.

The inter-query step is set to 1 sec—a shorter step was deemed unnecessary by end users. *WM* is longer than the step as the SDE stream is not temporally ordered, and ranges from 7 sec (2,450 SDEs) to 43 sec (15,050 SDEs). In the Helsinki infrastructure a 10 sec *WM* is sufficient, i.e. SDE delays are expected to be less than 10 sec. Figure 2(b) shows that we can achieve a significant performance gain by running RTEC in parallel. Such a gain is achieved without requiring SDE filtering. Moreover, Figure 2(b) shows that RTEC is capable of supporting real-time CE recognition at rush hour in Helsinki. For example, given a *WM* of 10 sec, RTEC recognises all CEs requested by end users in 260 ms when a single processor is used, and in 32 ms when all eight processors are used.

Larger datasets. In other applications, data velocity may be higher than that presented above. According to the results of the use case survey of the Event Processing Technical Society (EPTS) [7], in most applications there are at most 1,000 SDEs per sec. To test RTEC on higher SDE frequency, we generated datasets of 10,000 operating vehicles. In this case, we have 3,333 SDEs per sec coming from the 10,000 vehicles. We used all eight processors of the computer in parallel. Each instance of RTEC running on a processor performed recognition for 1,250 of these vehicles, and thus computed and stored the intervals of 25,000 CEs. Each processor received SDEs coming from *all* 10,000 vehicles. Figure 2(c) shows the average recognition times. As in the previous experiments, the step is set to 1 sec, while *WM* ranges from 7 to 43 sec. In this set-up, however, *WM* is ranging from 23,331 to 143,319 SDEs. Figure 2(c) shows that RTEC supports real-time recognition in this setting as well—RTEC recognises all CEs requested by end users in less than 1 sec.

Heterogeneous datasets. The experiments presented above were on homogeneous streams produced by buses and trams. The next set of experiments concerns heterogeneous streams from buses and Sydney Coordinated Adaptive Traffic System (SCATS) sensors, i.e. fixed sensors mounted on intersections. Our experiments were performed on real data for 1-31 January 2013, coming from the buses and SCATS sensors of Dublin, Ireland ⟨www.dublinked.ie⟩. The bus dataset covers 912 buses. Each operating bus emits a SDE every 20-30 sec with information about its position, delay and congestion. Information about in-vehicle temperature, noise and passenger density is not available in this dataset. On average, the bus dataset has a new SDE every 2 sec. The SCATS dataset includes 1,675 sensors, and each such

sensor transmits information every minute reporting on traffic flow and density. Given these streams, RTEC recognises CEs concerning bus punctuality, traffic congestion, traffic flow and density trends (for traffic congestion forecasting), and source disagreement i.e. when buses and SCATS sensors provide conflicting information on traffic congestion. Source disagreement recognition aids sensor fault diagnosis and repair. The choice of CEs, and their definitions, were specified in collaboration with domain experts. The CE definitions for this dataset require more complex reasoning than the definitions of PRONTO. Both sets of definitions are available with the RTEC code.

Figure 2(d) presents the results of two sets of experiments. First, we used a single processor to perform CE recognition for all 912 buses and 1,675 SCATS sensors, computing and storing the intervals of 7,765 CEs. Second, we used four processors in parallel. We took advantage of the existing SCATS sensor geodivision in Dublin—central, north, south and west city—and, therefore, each RTEC instance performed CE recognition for the SCATS sensors of, and buses going through one of the four Dublin areas. In this case, each RTEC instance computed and stored the intervals of approximately 2,169 CEs. As in the previous experiments, there was no SDE filtering to restrict the input relevant for each processor.

The inter-query step is set to 1 minute to coincide with the SCATS sensor update frequency. *WM* ranges from 10 to 130 minutes (32,000 to 435,000 SDEs). Figure 2(d) shows that RTEC is capable of supporting real-time CE recognition. E.g. given a 110 minute (368,000 SDEs) *WM*, RTEC performs CE recognition in 2 sec when four processors are used in parallel. Figure 2(d) also shows the performance gain of parallel recognition. The gain would have been more visible had there been a greater difference between the number of CEs in the single processor setting (7,765) and the distributed setting (2,169 per processor).

5.2 Public Space Surveillance

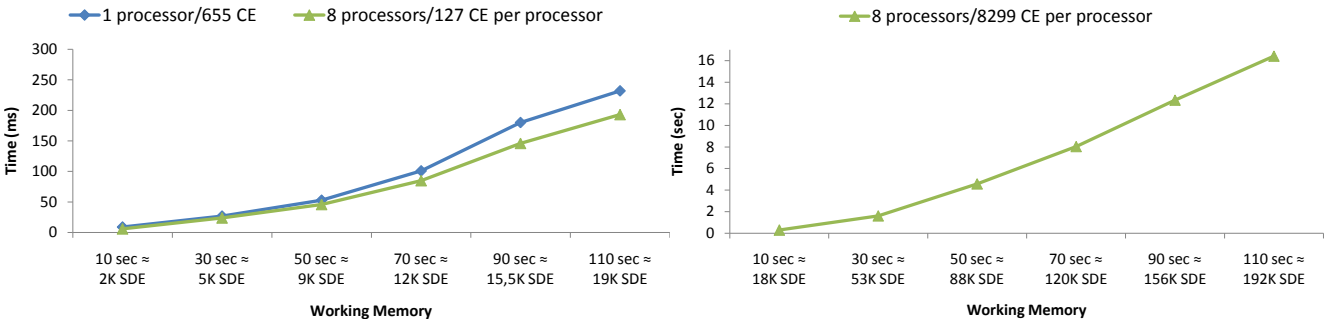
The second application concerns public space surveillance (PSS) from video content. We use the CAVIAR benchmark dataset consisting of 28 surveillance videos of a public space ⟨http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1⟩. The videos are staged—actors walk around, sit down, meet one another, leave objects behind, etc. Each video has been manually annotated by the CAVIAR team in order to provide the ground truth for 'short-term activities', i.e. activities taking place in a short period of time detected on individual video frames. (The frame rate in CAVIAR is 40 ms.) The short-term activities of CAVIAR concern an entity (person or object) entering or exiting the surveillance area, walking, running, moving abruptly, being active or inactive. The CAVIAR team has also annotated the 28 videos with 'long-term activities': a person leaving an object unattended, two people meeting, moving together and fighting. Short-term activities can be viewed as SDEs while long-term activities can be viewed as CEs. Consequently, the input to RTEC in this case study includes the set of annotated short-term activities, and the output is a set of recognised long-term activities.

'Forget' mechanism. The cost of the 'forget' mechanism depends on the *WM* size and the inter-query step size. In this application, we set $WM = Q_i - Q_{i-1}$ — in CAVIAR, SDEs are temporally sorted. Moreover, RTEC does not have to go through the complete list of SDEs available at each query time in order to decide which to retract. Whenever it can be safely assumed that the SDE stream is temporally sorted, the 'forget' mechanism can stop processing SDEs as soon as it finds the first such fact that starts after $Q_i - WM$. The performance gains can be significant. E.g. for $WM = Q_i - Q_{i-1} =$ 5,000 SDEs, the average 'forget' time is reduced from 27 ms to 17 ms.

CE recognition for multiple pairs of entities. Figure 3(a) shows the results of experiments concerning all 45 pairs of the 10 entities tracked in the CAVIAR dataset. (In CAVIAR each CE concerns a pair of entities.) On average, 179 SDEs are detected per sec. First, we used a single processor for CE recognition concerning all 45 tracked pairs. That requires computing and storing the intervals of 655 CEs. Second, we used all eight processors in parallel. In this case, each RTEC instance performed CE recognition concerning at most 6 entity pairs, computing and storing the intervals of 127 CEs. In both settings there was no SDE filtering. We varied *WM* from 10 sec (≈1,793 SDEs) to 110 sec (≈19,201 SDEs). As mentioned earlier, the inter-query step is equal to *WM*. Note that a simultaneous and equal increase of *WM* and inter-query step results in a more rapid increase of the total RTEC time than an increase of *WM* or step size separately (see e.g. Figure 2(b)). In all settings shown in Figure 3(a), RTEC performs real-time CE recognition.

In PSS, RTEC has to compute the SDE intervals from the given time-points at which a SDE takes place. In CTM in contrast the SDE intervals were computed by the SDE detection system and were made available when an interval ended. This task makes CE recognition in PSS more time-consuming. On the other hand, it may allow some CEs to be recognised at an earlier query time, as one does not have to wait for an SDE interval to end before recognising a CE.

Figure 3(a) shows that no significant performance gain is obtained by running RTEC in parallel on different processors. In relatively small CE numbers, such as those of CAVIAR, the difference of a few hundred CEs—in the centralised setting the single processor recognises only a few hundred more CEs than each of the eight processors in the distributed setting—affects only very slightly the performance of RTEC. Furthermore, there is some duplication of computation when we distribute recognition to various



(a) CE recognition for all 10 CAVIAR tracked entities.

(b) CE recognition for 100 tracked entities.

Fig. 3: Event Recognition for Public Space Surveillance.

processors. Consider e.g. that CE recognition for the pair (id_1, id_2) is performed on one processor while CE recognition for the pair (id_2, id_3) is performed on another processor. In both processors we will have to compute the intervals for which id_2 is a person, walking and stays inactive.

Larger datasets. We constructed a larger dataset by taking ten copies of the original CAVIAR dataset with new identifiers for the tracked entities in each copy. The resulting dataset has 100 tracked entities, i.e. 4,950 entity pairs, while on average 1,793 SDEs take place per sec. According to the EPTS use case survey [7], in the resulting dataset there are more SDEs per sec than in most applications. We used all eight processors of the computer in parallel. Consequently, each instance of RTEC running on a processor performed CE recognition for 619 entity pairs, computing and storing the intervals of 8,299 CEs. As in the previous set of experiments, there is no SDE filtering, and the inter-query step is equal to *WM*. We varied the sizes of the inter-query step and *WM* from 10 sec (≈17,933 SDEs) to 110 sec (≈192,010 SDEs). Figure 3(b) shows the average CE recognition times. In all cases RTEC performs real-time CE recognition.

6 Related Work

RTEC has a formal, declarative semantics as opposed to most complex event processing languages, several data stream processing and event query languages, and most commercial production rule systems. Furthermore, RTEC supports atemporal reasoning and reasoning over background knowledge, has built-in axioms for complex temporal phenomena, explicitly represents CE intervals and thus avoids the related logical problems, and supports out-of-order SDE streams. Concerning the Event Calculus literature, a key feature of RTEC is that it includes a windowing technique. In contrast, no Event Calculus system 'forgets' or represents concisely the SDE history.

One of the best known formal CE recognition systems is the Chronicle Recognition System (CRS) [15]. A 'chronicle' can be seen as a CE—it is expressed as a set of events linked together by time and context constraints. CRS has proven efficient and scalable enough for various application domains. However, CRS is a purely temporal reasoning system and cannot be directly used in applications requiring any type of atemporal reasoning, such as spatial reasoning.

In our approach to CE recognition, the availability of the power of logic programming is one of the main attractions of employing the Event Calculus. It allows CE definitions to include not only complex temporal constraints but also complex atemporal constraints. Furthermore, it allows reasoning over background knowledge. This is in contrast to various CE recognition approaches, such as [3], [15], [20], [11], that lack the ability of (complex) reasoning over existing domain knowledge [2]. The benefits of logic programming over other CE recognition approaches are reported in [28], while in [11] there is a detailed account of the limitations of existing approaches.

Shet et al. [32] have presented a logic programming approach to CE recognition from video content. Anicic et al. [2] have also developed a logic programming approach to CE recognition, and applied it to sensor networks. A distinguishing feature of our approach with respect to such lines of work is the use of an Event Calculus dialect for temporal representation and reasoning. RTEC has built-in axioms for temporal phenomena, including the formalisation of inertia, which facilitate considerably the development of succinct CE definitions, and therefore code maintenance.

Shanahan [31] formulated in first-order logic the 'simple', 'full' and 'extended' Event Calculus. Miller and Shanahan [25] translated a fragment of Shanahan's Event Calculus dialects to the language $\mathcal{E}$ [18] in order to make use of its software tools. $\mathcal{E}$, however, does not explicitly represent durative CEs. Other temporal formalisms that do not explicitly represent CE intervals may be found in [24], [15], [11]. The logical problems that arise from the lack of CE interval representation are reported in [27].

The 'Macro-Event Calculus' [8] includes axioms for computing the maximal intervals of simple fluents and extends the notion of 'macro-event' to support composite (macro) event operators: sequence, disjunction, parallelism and iteration. However, this dialect

cannot be used for run-time CE recognition as it does not have a mechanism for efficient reasoning over large datasets. The 'Interval-based Event Calculus' [27], [28] includes event operators for sequence, disjunction, mutual exclusivity, conjunction, concurrency, negation, quantification and aperiodicity. Event intervals in the Interval-based Event Calculus are closed: it is not possible to recognise a CE that started taking place at some earlier time-point and is still taking place. In many application areas, including CTM and PSS, this is a serious limitation.

The interval manipulation constructs of RTEC were developed primarily for efficiency reasons, and proved sufficient for CTM and PSS. The availability of logic programming allows us to express in RTEC various event operators, such as those of the Macro-Event Calculus and the Interval-based Event Calculus.

The Interval-based Event Calculus has a form of caching of fluents (only concerning time-points as fluent intervals are not represented) and event intervals. However, the possibilities of SDEs arriving in non-chronological order, and SDE revision, are not considered. Thus, in the Interval-based Event Calculus it is not possible e.g. to update the intervals of recognised CEs due to SDEs arriving with a delay. Note that several other event processing systems, such as [17], [14], [11], [13], [21], operate only under the assumption that SDEs are temporally sorted. Such systems rely on components that order SDEs prior to feeding them to the CE recognition system. RTEC does not rely on such components and may dynamically update the intervals of recognised CEs, or recognise new CEs, as a result of delayed SDE arrival or SDE revision. The applications mentioned in [10], [22], as well as CTM in the Helsinki and Dublin infrastructures, are but a few examples in which the SDE streams cannot be assumed to be ordered and/or may be revised.

The 'Cached Event Calculus' [10] performs *update-time* reasoning: it computes and stores the consequences of a SDE as soon as it arrives. Query processing, therefore, amounts to retrieving the appropriate CE intervals from the computer memory. The Cached Event Calculus does not 'forget' any SDE and has no technique for representing concisely the SDE history. Consequently, this dialect is not suitable for run-time CE recognition.

There are also important differences in the caching mechanism itself. In the Cached Event Calculus, when a maximal interval of a fluent is asserted or retracted due to a delayed SDE, the assertion/retraction is propagated to the fluents whose validity may rely on such an interval. E.g. $propagateAssert([T_1, T_2], U)$ in the Cached Event Calculus checks whether there are new initiations as a result of asserting the interval $(T_1, T_2]$ of fluent U. In particular, $propagateAssert$ checks whether: (1) the asserted fluent U is a condition for the initiation of a fluent F at the occurrence of event E, (2) the occurrence time T of E belongs to $(T_1, T_2]$, and (3) there is not already a maximal interval for F with T as its starting point. If the above conditions are satisfied, $propagateAssert$ recursively calls $updateInit(E, T, F)$ in order to determine if F is now initiated at T, and if it is, to update the fluent interval database accordingly.

$propagateAssert$ also checks whether there are new terminations as a result of a fluent interval assertion, while $propagateRetract$ checks whether there are new initiations and terminations as a result of a fluent interval retraction.

The cost of $propagateAssert$ and $propagateRetract$ is very high, especially in applications where the CE definitions include many rules with several fluents that depend on several other fluents. Furthermore, this type of reasoning is performed very frequently: $propagateAssert$ e.g. is invoked *every time* a fluent interval is asserted in the computer memory. If SDE revision were to be supported, then the number of invocations of $propagateAssert$ and $propagateRetract$ would increase. Note that the cost of such reasoning modules applies to all update-time reasoning approaches (e.g. [2]).

RTEC avoids the costly checks every time a fluent interval is asserted/retracted due to delayed SDE arrival/revision. We found that in RTEC it is more efficient, and simpler, to discard at each query time Q_i, all intervals of fluents representing CEs in $(Q_i - WM, Q_i]$ and compute from scratch all such intervals given the SDEs available at Q_i and detected in $(Q_i - WM, Q_i]$.

The 'Reactive Event Calculus' [9], [26] is another dialect performing update-time reasoning. This dialect is primarily developed to deal with ordered SDE streams. Furthermore, like the Cached Event Calculus, it does not 'forget' SDEs and has no technique for representing concisely the SDE history.

7 Summary and Further Work

We presented RTEC, an Event Calculus dialect with novel implementation techniques that allow for efficient CE recognition, scalable to large numbers of SDEs and CEs. A form of caching stores the results of sub-computations in the computer memory to avoid unnecessary recomputations. A set of interval manipulation constructs simplify CE definitions and improve reasoning efficiency. A simple indexing mechanism makes RTEC robust to SDEs that are irrelevant to the CEs we want to recognise and so can operate without SDE filtering modules. Finally, a 'windowing' mechanism supports real-time CE recognition. RTEC remains efficient and scalable in applications where SDEs arrive with a (variable) delay from, or are revised by, the SDE detection systems: it can update the already recognised CEs, and recognise new CEs, when SDEs are arrive with a delay or following revision.

The CE definitions on which our evaluation was performed are rather complex, taking into consideration SDEs as well as their context. Furthermore,

our CE definitions are significantly more complex than those presented in most related papers. Our empirical analysis showed that RTEC supports real-time CE recognition for very big city transport and traffic management, and large-scale surveillance applications. Moreover, it showed that RTEC is capable of meeting the performance requirements of most of today's applications, as estimated by an event processing use case survey [7].

We have not conducted comparisons of performance with other systems. We have described in detail the differences between our approach and a wide range of systems, and indicated why an empirical comparison might not be informative. There would also be the problem of compiling a new dataset that uses only features common to several systems. We are intending to look into the possibility of constructing such a dataset as part of our future work.

The manual construction of CE definitions is a time-consuming and error-prone process. We are developing techniques, based on abductive-inductive logic programming, for automated generation and refinement of CE definitions from very large datasets. We are also porting RTEC into probabilistic logic programming frameworks, in order to deal with various types of uncertainty, such as imperfect CE definitions, incomplete and erroneous SDE streams.

ACKNOWLEDGMENTS

This work has been partly funded by the EU FP7 projects PRONTO (231738) and SPEEDD (619435).

REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

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

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] A. Artikis, M. Sergot, and G. Paliouras. Run-time composite event recognition. In *DEBS*, pages 69–80. ACM, 2012.

[5] A. Artikis, A. Skarlatidis, F. Portet, and G. Paliouras. Logic-based event recognition. *Knowledge Engineering Review*, 27(4):469–506, 2012.

[6] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, 2006.

[7] 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.

[8] I. Cervesato and A. Montanari. A calculus of macro-events: Progress report. In *TIME*, pages 47–58, 2000.

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

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

[11] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *DEBS*, pages 50–61, 2010.

[12] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):15, 2012.

[13] N. Dindar, P. M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *DEBS*, pages 243–254, 2011.

[14] L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, pages 676–685, 2008.

[15] C. Dousson and P. Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchisation. In *IJCAI*, pages 324–329, 2007.

[16] 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.

[17] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex event processing over streams. In *CIDR*, 2007.

[18] A. Kakas, L. Michael, and R. Miller. Modular-$\mathcal{E}$: An elaboration tolerant approach to the ramification and qualification problems. In *LPNMR*, pages 211–226, 2005.

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

[20] 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.

[21] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *DEBS*, pages 291–302, 2011.

[22] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[23] D. Luckham and R. Schulte. Event processing glossary — version 1.1. Event Processing Technical Society, July 2008.

[24] K. Mahbub, G. Spanoudakis, and A. Zisman. A monitoring approach for runtime service discovery. *Automated Software Engineering*, 18(2):117–161, 2011.

[25] R. Miller and M. Shanahan. Some alternative formulations of the event calculus. In *Computational Logic: Logic Programming and Beyond*, LNAI 2408, pages 452–490. 2002.

[26] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst. Monitoring business constraints with the Event Calculus. *ACM TIST*, 5(1), 2014.

[27] A. Paschke. ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. Technical Report 11, Technische Universität München, 2005.

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

[29] A. Paschke and A. Kozlenkov. Rule-based event processing and reaction rules. In *Proceedings of RuleML*, LNCS 5858. 2009.

[30] T. Przymusinski. On the declarate semantics of stratified deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan, 1987.

[31] M. Shanahan. The event calculus explained. In *Artificial Intelligence Today*, LNAI 1600, pages 409–430. Springer, 1999.

[32] V. Shet, J. Neumann, V. Ramesh, and L. Davis. Bilattice-based logical reasoning for human detection. In *CVPR*, 2007.