Self-Adaptive Event Recognition for Intelligent Transport Management

Alexander Artikis[1], Matthias Weidlich[2], Avigdor Gal[2], Vana Kalogeraki[3] and Dimitrios Gunopulos[4]
[1]*Institute of Informatics & Telecommunications, NCSR Demokritos, Athens, Greece, a.artikis@iit.demokritos.gr*
[2]*Technion - Israel Institute of Technology, Haifa, Israel, {weidlich@tx, avigal@ie}.technion.ac.il*
[3]*Department Informatics, Athens University of Economics and Business, Greece, vana@aueb.gr*
[4]*Department of Informatics and Telecommunications, University of Athens, Greece, dg@di.uoa.gr*

***Abstract*—Intelligent transport management involves the use of voluminous amounts of uncertain sensor data to identify and effectively manage issues of congestion and quality of service. In particular, urban traffic has been in the eye of the storm for many years now and gathers increasing interest as cities become bigger, crowded, and "smart". In this work we tackle the issue of uncertainty in transportation systems stream reporting. The variety of existing data sources opens new opportunities for testing the validity of sensor reports and self-adapting the recognition of complex events as a result. We report on the use of a logic-based event reasoning tool to identify regions of uncertainty within a stream and demonstrate our method with a real-world use-case from the city of Dublin. Our empirical analysis shows the feasibility of the approach when dealing with voluminous and highly uncertain streams.**

***Keywords*-event processing; pattern matching; event calculus**

I. Introduction

Detecting complex event patterns from multiple, highly uncertain data streams is a promising vehicle to support Big Data applications for monitoring, detection, and online response [4], [9]. Consider, for example, an urban monitoring system that identifies road congestions and responds by applying local changes to traffic light control policies to reduce ripple effects. Such a system may use events that report on the flow in junctions together with reports from buses to collect evidence of congestions in-the-make.

Two of the main challenges when dealing with Big Data are that of *variety* and *veracity*. Data, arriving from multiple heterogeneous sources, may be of poor quality and in general requires pre-processing and cleaning when used for analytics and query answering. In particular, sensor networks introduce uncertainty into the system due to reasons that range from inaccurate measurements through network local failures to unexpected interference of mediators. While the first two reasons are well recorded in the literature, the latter is a new phenomenon that stems from the combination of variety and sensor data. Sensor data may go through multiple mediators en route to our systems. Such mediators apply various filtering and aggregation mechanisms, most of which are unknown to the system that receives the data. Hence, the uncertainty that is inherent to sensor data is multiplied by the factor of unknown aggregation and filtering treatments.

In this work we outline the principle of using *variety* to effectively handle *veracity*. In a nutshell, streams from multiple sources are used to generate common complex events. These events are matched against each other to identify mismatches that indicate uncertainty regarding the event streams. Temporal regions of uncertainty are identified from which point the monitoring system autonomously decides on how to manage this uncertainty. At times, complete event intervals are neglected. At other times, a selection mechanisms prefers one stream over the others. Finally, using multiple sources, one can create a distribution over the possible occurrence of events in inconsistent regions.

Our tool of choice for this task is the RTEC (Run-Time Event Calculus) event recognition engine [2]. In addition to standard event algebra operators, RTEC has a built-in representation of the law of inertia [7] that makes it particularly useful for expressing rules that dynamically discard noisy event sources and include reliable ones.

We illustrate our approach using real, heterogeneous data streams concerning city transport and traffic management. First, we use data from Sydney Coordinated Adaptive Traffic System (SCATS) sensors, that is, fixed sensors mounted on intersections to measure traffic flow. Second, we use bus probe data stating, among others, the location, line and delay of each bus as well as traffic congestions. The voluminous data streams come from the city of Dublin, Ireland, and concern all SCATS sensors of the city and the complete bus fleet. To the best of our knowledge, this is the first approach combining these heterogeneous streams for real-time intelligent transport management.

The contributions of the paper are summarized as follows:

- At a conceptual level, we show how cross-validating multiple data streams can be used for self-adaptive event processing that enhances stream credibility.
- We show how the use of semantics can support reasoning for such cross-validation.
- We provide empirical evidence to the feasibility of the proposed approach.

Organisation. Section II provides an introduction to complex event processing and discusses related research on uncertain data stream handling. Section III presents the event recognition engine that we use. Section IV demonstrates how to model event patterns for city transport and traffic management. These patterns are used in Section V to demonstrate self-adaptation for noisy data stream handling.

Section VI presents our empirical evaluation, showing the method feasibility, while Section VII summarises our work.

II. Background and Related Work

Event processing refers to an approach to software systems that is based on event delivery, and that includes specific logic to filter, transform, or detect patterns in events as they occur. Event processing platforms are diversified into products with various approaches towards event processing, including the stream oriented approach, the rule oriented approach, the imperative approach, and the publish-subscribe approach. As a common denominator, all of these approaches assume that all relevant events are consumed by the event processing system, all events reported to the system have occurred, and event processing can be done in a deterministic fashion.

Etzion and Niblett [4] discuss the roots of uncertainty in an event-based system and present several approaches to handle it. Gal et al. [5] model uncertainty in data steams and survey techniques to manage it, focusing on uncertainty of rules. Generally speaking, there may be four different approaches to deal with event uncertainty. First, uncertainty can simply be ignored. Such a solution may be cost-effective if uncertainty is relatively infrequent, and the damage of not handling it is not substantial. A second approach requires complex events to be recognized only when the event pattern is a necessary and sufficient condition to deterministic detection. Most event processing systems follow these two approaches, that is, they do not manage uncertainty [3].

According to a third approach, it is possible to notify the result of detected events, along with an indication (such as probability) to its validity. Methods for this approach are based on probabilistic graphical models [13], Markov Logic Networks [12], probabilistic logic programming [11], and fuzzy set and possibility theory [8]. Although there is considerable work on optimising probabilistic reasoning techniques, the imposed overhead does not allow for real-time performance in a wide range of applications [1].

Finally, event recognition can be designed so that it makes use of reinforcement from multiple indications. Our work falls into this category, in which multiple sources are used for cross-validation. Due to volume and velocity, cleaning the data in advance (see, for example, [6]) is not always feasible, while common uncertainty elimination techniques, such as load shedding based on confidence, are in most cases ineffective [1].

These last two approaches are orthogonal and systems may use one or both approaches. While our proposed method falls under the fourth approach, its outcome can be used to support other approaches as well. For example, by identifying temporal regions of inconsistency, one can establish an empirical probability distribution over the event streams. Also, discarding inconsistent streams effectively creates a deterministic dataset, as is promoted by the second approach.

III. A Logic-based Event Model

We present a logic-based event processing model based on the Event Calculus for Run-Time reasoning (RTEC) [2]. The Event Calculus, introduced in [7], is a logic programming formalism for reasoning about events and their effects. Based on [2], we summarise the essentials of the model. We adopt the common logic programming convention that variables start with upper-case letters (and are universally quantified, unless otherwise indicated), while predicates and constants start with lower-case letters. Our approach relies on logic programming due to the formal, declarative semantics and the rich expressiveness it offers. RTEC supports efficient reasoning (as is evident in the empirical evaluation), and thus serves us well in illustrating the approach feasibility.

A. Event Representation

Systems for event recognition (event pattern matching [9]) accept as input a stream of time-stamped simple, derived events (SDE). An SDE (or low-level event) is the result of applying a computational derivation process to some other event, such as an event coming from a sensor [10]. Events that arrive at the system are not the raw events emitted by sensors. Such raw events are enriched, filtered, and aggregated by multiple mediators, whose internal functionalities may not be known to the event recognition system. As indicated above, such preprocessing may result in uncertainty as to the validity of the events, and the use of multiple event streams for cross-validation is at the heart of our proposed solution.

Using SDE as input, event recognition systems identify composite events (CE) of interest — collections of events that satisfy some pattern. The specification of a CE (or high-level event) imposes temporal and, possibly, atemporal constraints on its deriving events, either SDEs or other CEs.

In the RTEC model, types of events are represented as n-ary predicates *event(Attribute1,...,AttributeN)*, such that the parameters define the attribute values of an event instance *event(value1,...,valueN)*. An example from the Dublin traffic management scenario is the type of SDE emitted by SCATS sensors, *traffic(StreetSegId, Flow, Count)*, which refers to the measured traffic flow and aggregate number of vehicles passing some sensor (identified by the attribute *StreetSegId*). Thus, a specific event instance is an instantiation of this predicate with constant values, *e.g.*, *traffic(s187, 4.51, 117)*.

Time is assumed to be linear and discrete, represented by integer time-points. The occurrence of an event E at time T is modelled by the two-ary predicate $\mathsf{happensAt}(E, T)$. To reason about the effects of events, we rely on the notion of a *fluent* F, a property that is allowed to have different values at different points in time. Here, the term $F = V$ denotes that fluent F has value V. Informally, $\mathsf{holdsAt}(F = V,\ T)$ represents that fluent F has value V at a particular time-point T. Interval-based semantics are obtained with the predicate $\mathsf{holdsFor}(F = V,\ I)$, where I is a list of maximal intervals for which fluent F has value V continuously. holdsAt and

Table I
RTEC PREDICATES.

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
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' wrt every list of maximal intervals of list L

holdsFor are defined in such a way that, for any fluent F, holdsAt$(F=V,\ T)$ if and only if time-point T belongs to one of the maximal intervals of I for which holdsFor$(F=V,\ I)$. Table I presents the main RTEC predicates.

Fluents are of two kinds: *simple* and *statically determined*. For a simple fluent F, $F=V$ holds at a particular time-point T if $F=V$ has been *initiated* by an event at some time-point earlier than T (using predicate initiatedAt), and has not been *terminated* in the meantime (using predicate terminatedAt). This is an implementation of the *law of inertia*. Statically determined fluents are defined by means of interval manipulation constructs, such as union_all, intersect_all and relative_complement_all (see Table I).

In our model, the input SDE streams are represented by logical facts that define event instances (predicate happensAt) or the values of fluents (predicates holdsAt and holdsFor). Taking up the example of the SCATS sensor given earlier, facts of the following structure model the input stream:

$$\text{happensAt}(\mathit{traffic}(\mathit{StreetSegId}, \mathit{Flow}, \mathit{Count}),\ T)$$

CEs, in turn, are modelled as logical rules defined over event instances (happensAt), the effects of events (initiatedAt and terminatedAt), or the values of the fluents (holdsAt and holdsFor), and implement the respective temporal and atemporal constraints. For illustration, consider a CE that captures whether traffic flow as given by *traffic* SDE is decreasing. We may capture the CE as a simple fluent *flowTrend* that assumes the value *decreasing* if in two consecutive SDEs (the second one occurring six minutes, that is, 360×10^6 milliseconds, after the first one) there is a drop of more than 10% in the flow value:

$$\begin{aligned}&\text{initiatedAt}(\mathit{flowTrend}(S)=\mathit{decreasing},\ T) \leftarrow\\ &\quad \text{happensAt}(\mathit{traffic}(S, \mathit{Flow}, \_),\ T),\\ &\quad \text{happensAt}(\mathit{traffic}(S, \mathit{Flow}', \_),\ T+360\times10^6),\\ &\quad \mathit{Flow}' < V-\mathit{Flow}\times 0.1\end{aligned} \tag{1}$$

'_' denotes a 'free' variable that is not bound in a rule.

B. Run-Time Composite Event Recognition

Based on the introduced model, run-time CE recognition is performed as follows. The RTEC engine queries, computes and stores 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 SDEs that fall within a specified interval — the 'working memory' (*WM*) or 'window' — are taken into consideration: all SDEs that took place before or on Q_i-WM are discarded. This constraint ensures that the cost of CE recognition depends only on the size of *WM* and not on the complete SDE history. The size of *WM*, as well as the temporal distance between two consecutive query times — the 'step' (Q_i-Q_{i-1}) — are tuning parameters that can be either chosen by the user or optimized for performance.

The relationships between *WM* and Q_i-Q_{i-1} can be divided into three cases, as follows.

- $WM < Q_i-Q_{i-1}$, 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 SDEs arrive at the engine in a timely manner. If SDEs do not arrive in a timely manner, then the effects of SDEs that took place before Q_i but arrived after Q_i will be lost.
- $WM > Q_i-Q_{i-1}$. In the common case that SDEs arrive at the engine with delays, it is preferable to make *WM* longer than the step. This way, it becomes possible to compute, at Q_i, the effects of SDE that took place in $(Q_i-WM, Q_{i-1}]$, but arrived after Q_{i-1}.

Further details on RTEC may be found in [2].

IV. EVENT RECOGNITION FOR TRANSPORT & TRAFFIC MANAGEMENT

The model presented above is used for city transport and traffic management in the city of Dublin. We first describe the input event streams of the different sources (Section IV-A). Then, we turn to the definition of CE in Section IV-B.

A. Input Event Streams

To guide transport and traffic management, the input to the event recognition engine consists of SDE that come from two heterogeneous data streams with different time granularity. First, static sensors mounted on various junctions (SCATS sensors) transmit every six minutes information about traffic flow and the aggregate number of vehicles passing over. In addition, buses transmit information about their position and congestions every 20-30 sec. Buses transmit SDEs with the following structures:

$$\text{happensAt}(\mathit{move}(\mathit{Bus}, \mathit{Line}, \mathit{Operator}, \mathit{Delay}),\ T)$$
$$\text{holdsAt}(\mathit{gps}(\mathit{Bus}, \mathit{Lon}, \mathit{Lat}, \mathit{Direction}, \mathit{Congestion})=\text{true},\ T)$$

$move(Bus, Line, Operator, Delay)$ records an event where Bus runs in $Line$ with a $Delay$ at time T (in microseconds), and is owned by $Operator$. $Delay$ is a possibly negative integer measured in seconds. In addition, $gps(Bus, Lon, Lat, Direction, Congestion)$ states the longitude and latitude location of the Bus, as well as its direction (0 or 1) on the $Line$. Further, the gps fluent provides information about congestion (0 or 1) in the given location.

B. Composite Events

Several CEs can be derived from the input streams and reported to the city transport decision-makers. These CEs relate to, among others, bus driving quality, traffic flow (trends), as well as traffic congestion. The choice of CE, and their definitions, were specified in collaboration with the domain experts. In this section, we focus on traffic congestion. Information on congestion is reported in both input streams, SCATS sensors and buses. Consider first congestion as reported by SCATS sensors:

$$\begin{aligned}&\mathsf{holdsFor}(scatsReportedCongestion(Lon_S, Lat_S) = \mathsf{true}, I) \leftarrow \\ &\quad \mathsf{holdsFor}(flow(Lon_S, Lat_S) = low,\ I_1), \\ &\quad \mathsf{holdsFor}(aggregateCount(Lon_S, Lat_S) = low,\ I_2), \\ &\quad \mathsf{intersect\_all}([I_1, I_2],\ I)\end{aligned} \tag{2}$$

$flow(Lon_S, Lat_S)$ and $aggregateCount(Lon_S, Lat_S)$ are simple fluents classifying the traffic flow and aggregate number of vehicles passing over the SCATS sensor at location (Lon_S, Lat_S) as low/average/high. Whether the traffic flow and aggregate count is low (respectively average/high) at some SCATS sensor depends on the capacity of the road in which the sensor is installed. intersect_all computes the intersection of a list of lists of maximal intervals — see Table I for the interval manipulation constructs of RTEC. According to rule (2), a congestion is said to be reported at the location (Lon_S, Lat_S) of a SCATS sensor if the traffic flow and the aggregate number of vehicles passing through that location are low.

The congestion definition presented above is sufficient for the given SDE, as these are defined in the SCATS dataset. Alternative congestion definitions from fixed sensors are beyond the scope of this paper.

Congestion is also reported by buses — this is very useful as there are numerous areas in the city that do not have SCATS sensors. Consider the following formalisation:

$$\begin{aligned}&\mathsf{initiatedAt}(busReportedCongestion(Lon, Lat) = \mathsf{true},\ T) \leftarrow \\ &\quad \mathsf{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\quad \mathsf{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 1),\ T), \\ &\quad close(Lon_B, Lat_B, Lon, Lat) \\ &\mathsf{terminatedAt}(busReportedCongestion(Lon, Lat) = \mathsf{true}, T) \leftarrow \\ &\quad \mathsf{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\quad \mathsf{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 0),\ T), \\ &\quad close(Lon_B, Lat_B, Lon, Lat)\end{aligned} \tag{3}$$

(Lon, Lat) are the coordinates of some area of interest, while (Lon_B, Lat_B) are the current coordinates of a Bus. The gps fluent, like the $move$ event, is given by the dataset. $close$ is an atemporal predicate computing the distance between two points and comparing them against a threshold. $busReportedCongestion(Lon, Lat)$ starts being true when a bus moves close to the location (Lon, Lat) for which we are interested in detecting congestions, and (the bus) reports a congestion (represented by 1 in the gps fluent). Moreover, $busReportedCongestion(Lon, Lat)$ stops being true when a (possibly different) bus moves close to (Lon, Lat) and reports no congestion (represented by 0 in gps).

V. Self-Adaptation

Event processing applications deal with various types of uncertainty such as incomplete and erroneous SDE streams [1]. Not surprisingly, the bus and SCATS datasets on which we perform CE recognition are no exception. Various SDE fields are incomplete or incorrect and sensors provide conflicting information. Due to the significant overhead of probabilistic reasoning, uncertainty is often ignored (see Section II), compromising by that CE recognition accuracy. In this section we present an approach for self-adaptive CE recognition that deals with some types of noise without using probabilistic reasoning techniques. Consequently, we may improve accuracy while maintaining real-time performance.

Our methodology is simple. It involves representing noise as CE, using the same mechanism that aims at detecting CE patterns to tell us whether our detection contains embedded uncertainty. By doing so, we can gain from the efficient mechanisms that were developed for event processing to recognize and manage uncertainty. Next, we provide several instances to this methodology.

We look into types of noise that are known in advance, either via machine learning or expert knowledge. For example, it is known that SCATS sensors that are close to each other often report conflicting traffic flow values, which we interpret as uncertainty due to the sensor proximity. We model such a noise type as a CE, and upon detection we adapt the CE recognition process by (temporarily) discarding the corresponding sensor(s). The following example expresses SCATS sensor flow noise:

$$\begin{aligned}&\mathsf{holdsFor}(sensorFlowNoise(S_1, S_2) = \mathsf{true},\ I) \leftarrow \\ &\quad \mathsf{holdsFor}(flowTrend(S_1) = increasing,\ I_1), \\ &\quad \mathsf{holdsFor}(flowTrend(S_2) = decreasing,\ I_2), \\ &\quad \mathsf{intersect\_all}([I_1, I_2],\ I_3), \\ &\quad \mathsf{holdsFor}(flowTrend(S_1) = decreasing,\ I_4), \\ &\quad \mathsf{holdsFor}(flowTrend(S_2) = increasing,\ I_5), \\ &\quad \mathsf{intersect\_all}([I_4, I_5],\ I_6),\ \mathsf{union\_all}([I_3, I_6],\ I)\end{aligned} \tag{4}$$

A partial definition of the $flowTrend$ fluent was given in rule (1), while union_all computes the union of a list of lists of maximal intervals (see Table I). The $sensorFlowNoise(S_1, S_2)$ CE is recognised when two

SCATS sensors S_1 and S_2 have conflicting traffic flow values, that is, the flow value in S_1 is increasing (respectively decreasing) while in S_2 is decreasing (increasing). We compute the maximal intervals for which $sensorFlowNoise(S_1, S_2) = \text{true}$ holds continuously only for sensors S_1 and S_2 that are close to each other.

In Dublin three sensors are often placed close to each other, all reporting on a specific junction. In case one of these sensors contradicts the others, the former may be temporarily discarded from the CE recognition process. An implementation of this idea leads to a variant of rule (2) that detects traffic congestion by means of SCATS sensors:

$$\begin{aligned}&\text{holdsFor}(scatsReportedCongestion(Lon_S, Lat_S) = \text{true}, I) \leftarrow \\ &\quad \text{holdsFor}(flow(Lon_S, Lat_S) = low,\ I_1), \\ &\quad \text{holdsFor}(aggregateCount(Lon_S, Lat_S) = low,\ I_2), \\ &\quad \text{intersect\_all}([I_1, I_2],\ I_3), \\ &\quad \text{holdsFor}(noisySCATS(S, Lon_S, Lat_S) = \text{true},\ I_4), \\ &\quad \text{relative\_complement\_all}(I_3,\ [I_4],\ I)\end{aligned} \tag{2'}$$

The maximal intervals for which $noisySCATS(S, Lon_S, Lat_S) = \text{true}$ are computed by the union of the lists of maximal intervals for which $sensorFlowNoise(S, S_1) = \text{true}$ and $sensorFlowNoise(S, S_2) = \text{true}$, where S_1 and S_2 are the SCATS sensors that are close to S. In $\text{relative\_complement\_all}(I', L, I)$, 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. According to rule (2′), a SCATS sensor S is not taken into consideration in the detection of a congestion as long as S has a conflicting flow trend with its near-by SCATS sensors.

We also identify the conditions in which a bus provides potentially noisy information in order to adapt the CE recognition process by (temporarily) discarding the information provided by the noisy bus. This idea is implemented as follows ('not' denotes negation-by-failure):

$$\begin{aligned}&\text{initiatedAt}(noisy(Bus) = \text{true},\ T) \leftarrow \\ &\ \text{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\ \text{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 1),\ T), \\ &\ close(Lon_B, Lat_B, Lon_S, Lat_S), \\ &\ \text{not holdsAt}(scatsReportedCongestion(Lon_S, Lat_S) = \text{true}, T) \\ &\text{terminatedAt}(noisy(Bus) = \text{true},\ T) \leftarrow \\ &\ \text{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\ \text{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 1),\ T), \\ &\ close(Lon_B, Lat_B, Lon_S, Lat_S), \\ &\ \text{holdsAt}(scatsReportedCongestion(Lon_S, Lat_S) = \text{true}, T) \\ &\text{terminatedAt}(noisy(Bus) = \text{true},\ T) \leftarrow \\ &\ \text{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\ \text{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 0),\ T), \\ &\ close(Lon_B, Lat_B, Lon_S, Lat_S), \\ &\ \text{not holdsAt}(scatsReportedCongestion(Lon_S, Lat_S) = \text{true}, T)\end{aligned} \tag{5}$$

A bus is said to provide noisy information about congestions if a SCATS sensor contradicts it, assuming here that SCATS sensors are more trustworthy than buses. More precisely, $noisy(Bus) = \text{true}$ is initiated when the Bus reports a congestion at its current location (Lon_B, Lat_B), the current location of the bus is close to the location (Lon_S, Lat_S) of SCATS sensor S, and S does not report a congestion. $noisy(Bus) = \text{true}$ is terminated when the Bus moves close to a SCATS sensor S and agrees with S on congestion.

This definition of $noisy(Bus)$ is designed to reduce false positives, that is, when a bus reports a congestion in an area in which the SCATS sensors do not detect a congestion. In other applications, we may additionally/alternatively aim at reducing false negatives.

Using $noisy(Bus)$ we adapt the $busReportedCongestion$ definition that reports congestion from bus data:

$$\begin{aligned}&\text{initiatedAt}(busReportedCongestion(Lon, Lat) = \text{true},\ T) \leftarrow \\ &\quad \text{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\quad \text{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 1),\ T), \\ &\quad \text{not holdsAt}(noisy(Bus) = \text{true}), \\ &\quad close(Lon_B, Lat_B, Lon, Lat) \\ &\text{terminatedAt}(busReportedCongestion(Lon, Lat) = \text{true},\ T) \leftarrow \\ &\quad \text{happensAt}(move(Bus, \_, \_, \_),\ T), \\ &\quad \text{holdsAt}(gps(Bus, Lon_B, Lat_B, \_, 0),\ T), \\ &\quad \text{not holdsAt}(noisy(Bus) = \text{true}), \\ &\quad close(Lon_B, Lat_B, Lon, Lat)\end{aligned} \tag{3'}$$

According to this new formalisation, the congestion information offered by a bus is discarded as long as the bus is contradicted by SCATS sensors.

The performance overhead of self-adaptive event recognition for dealing with noisy data streams involves mainly the recognition of additional CEs expressing various types of noise. The number of such CEs is proportional to the number of event sources. In most applications, this means that there will only be a small increase in the number of CEs being recognised. In the following section we evaluate the execution overhead on the city of Dublin.

VI. Empirical Evaluation

Experimental Setup. Our experiments were performed on real data streams coming from the buses and SCATS sensors of Dublin, Ireland. The streams were collected from February 1 to April 30 in 2012 and comprise 13GB of data. The bus dataset includes 942 buses and 91 lines/routes. Each operating bus emits SDEs every 20-30 seconds with information about its position, delay and congestions. On average, the bus dataset has a new SDE every 2 seconds. The SCATS dataset includes 966 sensors. Each SCATS sensor transmits information every six minutes and reports on traffic flow and aggregate number of vehicles passing over the sensor. Excerpts from both datasets are publicly available[1].

[1] http://www.dublinked.ie

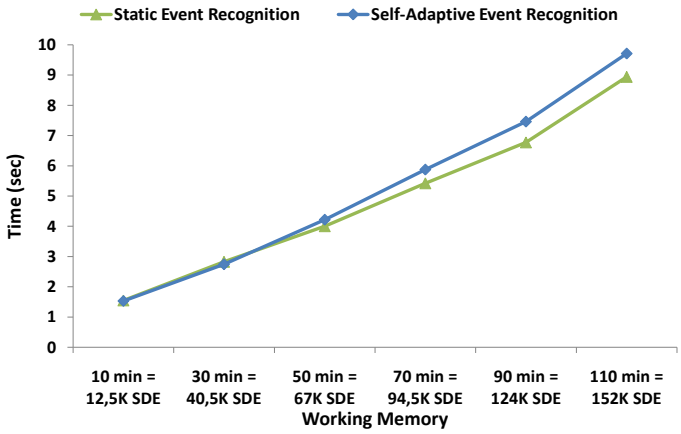

Figure 1. Static and self-adaptive CE recognition in a single processor.

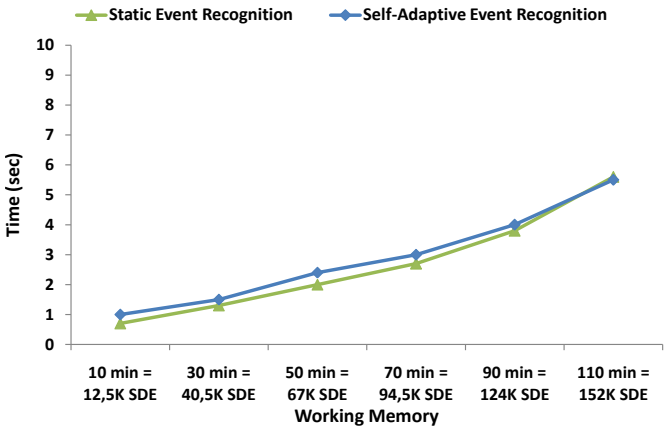

Figure 2. Static and self-adaptive CE recognition in 4 processors.

We recognise CEs concerning individual bus punctuality, line punctuality (the maximal intervals for which a line consisting of a number of buses is said to be punctual), bus driving quality, traffic flow (trends), and congestions (in any location of interest in Dublin). Additionally, to allow for self-adaptation, we compute the maximal intervals for which a SCATS sensor is contradicted by near-by SCATS sensors, and the maximal intervals for which a bus is contradicted by a SCATS sensor on congestion.

The experiments were run on a computer with eight Intel i7 950@3.07GHz processors and 12GB RAM, running Ubuntu Linux 12.04 and YAP Prolog 6.2.2.

Results. We first compare two sets of experiments. In the first, we performed 'static' recognition, that is, CE recognition that always takes into consideration all event sources. In this case, at each query time RTEC computes and stores the maximal intervals of around 51,000 CEs. Then, we performed self-adaptive event recognition where noisy sensors are detected at run-time and the system autonomously discards them until they resume offering reliable information. In this case, RTEC computes and stores the maximal intervals of 53,500 CEs with a 5% increase in the number of CEs compared to the static setting. Figure 1 displays the average CE recognition times in CPU seconds. The working memory (*WM*) ranges from 10 min, including on average 12,500 SDEs, to 110 minutes, including 152,000 SDEs. CE recognition was performed on a single processor.

Figure 1 shows that self-adaptive CE recognition has a minimal overhead compared to static recognition. The overhead is due to computing and storing the maximal intervals of additional CEs, capturing the intervals for which some sensors are considered unreliable.

Figure 1 also shows that RTEC performs real-time CE recognition both in the static and the self-adaptive setting. For example, in a 10 min *WM* both static and self-adaptive recognition take on average around 1 sec, while in a 110 min *WM* recognition takes on average around 9 sec.

CE recognition performance is not entirely dependent on the size of the input data streams. The complexity of the CE definitions affects significantly recognition times. In this work we make use of quite complex CE definitions including various constraints on multiple entities (buses running on the same line, for example) and heterogeneous data streams. This is in contrast to the vast majority of the event processing literature where quite simple CE definitions are used.

CE recognition for transport and traffic management, as defined here, is straightforward to distribute. In Dublin, for instance, SCATS sensors are placed into the intersections of four geographical areas: central city, north city, west city, and south city. We distributed CE recognition accordingly. We used four processors of the computer on which we performed the experiments — each processor computed CEs concerning the SCATS sensors of one of the four areas of Dublin as well as CE concerning the buses that go through that area.

Figure 2 shows the average CE recognition times for the static and the self-adaptive settings. We see that the performance difference between static and self-adaptive CE recognition is smaller than before. This is due to the fact that, in each processor, the difference in the number of CEs between the static and the self-adaptive setting is smaller as each processor computes CEs concerning a smaller number of event sources. For example, the processor responsible for the largest geographical area of Dublin computes and stores the maximal intervals of 18,000 CEs in the static setting and 19000 CEs in the self-adaptive setting. In other words, in distributed, self-adaptive CE recognition the intervals of at most 1,000 additional CEs are computed while in centralised, self-adaptive recognition we compute the intervals of 2,500 additional CEs.

Distributing CE recognition to four processors lead to significant performance gain. The input to each processor is restricted to the SDEs of the SCATS sensors and buses for which it performs CE recognition. Furthermore, as mentioned above, each processor has to compute and store the maximal

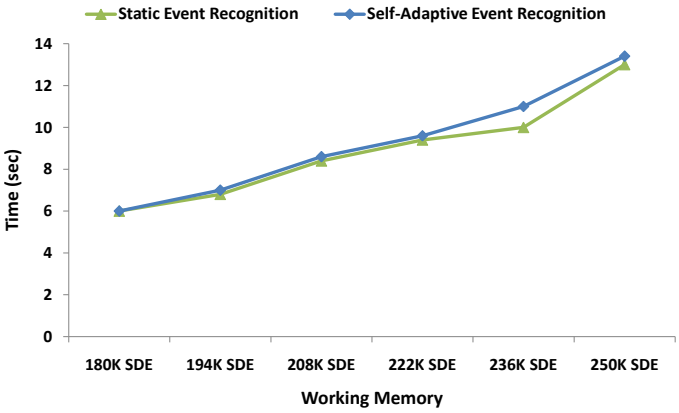

Figure 3. Static and self-adaptive CE recognition in 4 processors: high velocity data streams.

intervals of a smaller number of CEs. One may further distribute CE recognition by dividing further the geographical area of the city under consideration, thus reducing CE recognition times.

To test our approach on higher velocity data streams, we increased further the *WM* size. Figure 3 shows the average CE recognition times when *WM* ranges from 180,000 SDEs to 250,000 SDE. In these experiments four processors were used in parallel. As shown in Figure 3, the performance of self-adaptive CE recognition is very close to that of static recognition even in larger data streams. Therefore, we can conclude that data velocity has only minor impact on the performance overhead of self-adaptation.

VII. Summary and Future Work

Two of the main challenges when dealing with Big Data are those of *variety* and *veracity*. We presented the principle of using *variety* to effectively handle *veracity*. Streams from multiple sources are used to generate common composite events. These events are matched against each other to identify mismatches that indicate uncertainty regarding the event streams. Temporal regions of uncertainty are identified from which the system autonomously decides to adapt its event sources in order to deal with uncertainty, without compromising efficiency. At times, complete event intervals are neglected. At other times, a selection mechanisms prefers one stream, or an entity within a stream, over the others.

Our approach is generic and applicable to any event processing application in which noise types can be determined via expert knowledge or machine learning techniques. For illustration purposes, we used real, heterogeneous data streams from SCATS sensors and buses in Dublin. To the best of our knowledge, this is the first approach combining these heterogeneous streams for real-time intelligent transport management. Our experiments show that self-adaptive event recognition compromises efficiency only very slightly, and therefore, is a promising approach to dealing with veracity in Big Data applications.

For future work, we intend to explore the increase in accuracy achieved by self-adaptive event recognition. In the INSIGHT project[2], we are collecting datasets for which ground truth is available (for instance, by means of tweets sent to the radio station reporting on congestions and quality of transport). In this way, we will be able to perform a balanced comparison with probabilistic reasoning approaches.

Acknowledgment

This work was supported by the EU INSIGHT project (FP7-ICT 318225) and the ERC IDEAS NGHCS project.

References

[1] A. Artikis, O. Etzion, Z. Feldman, and F. Fournier. Event processing under uncertainty. In *DEBS*, pages 32–43. ACM, 2012.

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

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

[4] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010.

[5] A. Gal, S. Wasserkrug, and O. Etzion. Event processing over uncertain data. In *Reasoning in Event-Based Distributed Systems*, pages 279–304. Springer, 2011.

[6] Shawn R. Jeffery, Minos N. Garofalakis, and Michael J. Franklin. Adaptive cleaning for rfid data streams. In *VLDB*, pages 163–174, 2006.

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

[8] H. Liu and H.-A. Jacobsen. Modeling uncertainties in publish/subscribe systems. In *ICDE*, pages 510–522, 2004.

[9] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

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

[11] A. Skarlatidis, A. Artikis, J. Filippou, and G. Paliouras. A probabilistic logic programming event calculus. *Theory and Practice of Logic Programming*, 2013.

[12] A. Skarlatidis, G. Paliouras, G. Vouros, and A. Artikis. Probabilistic event calculus based on markov logic networks. In *RuleML America*, pages 155–170, 2011.

[13] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Efficient processing of uncertain events in rule-based systems. *IEEE Trans. Knowl. Data Eng.*, 2011.

[2]http://www.insight-ict.eu