Being logical or going with the flow? A comparison of Complex Event Processing systems

Elias Alevizos and Alexander Artikis

National Centre for Scientific Research (NCSR) "Demokritos", Athens 15310, Greece
alevizos.elias@iit.demokritos.gr, a.artikis@iit.demokritos.gr

Abstract. Complex event processing (CEP) is a field that has drawn significant attention in the last years. CEP systems treat incoming information as flows of time-stamped events which may be structured according to some underlying pattern. Their goal is to extract in real-time those patterns or even learn the patterns which could lead to certain outcomes. Many CEP systems have already been implemented, sometimes with significantly different approaches as to how they represent and handle events. In this paper, we compare the widely used Esper system which employs a SQL-based language, and RTEC which is a dialect of the Event Calculus.

1 Introduction

As the number of possible sources of information which can feed a system with real-time data increases, so does the need for distributed systems with the ability to efficiently handle flows of data. The most typical scenario is one in which a network of sensors has been installed, with each sensor sending its readings to a (possibly distributed) processing system. The system's goal is to detect (or even learn) in real-time certain patterns present in the incoming data flows, so that the appropriate preventive or reinforcing action be taken. Domains in which such systems could prove helpful are network intrusion detection, traffic management and environmental monitoring, to name but a few.

The usual Database Management Systems (DBMS) have certain features, like the requirement for storing before processing or that of asynchronous processing, which prevent them from being directly transferred to the problem of stream processing and pattern matching. During the last decade, a significant number of the so-called complex event processing (CEP) systems have appeared that attempt to overcome the limitations of typical DBMSs [5], [6]. A CEP system attempts to inverse the human-active database-passive (HADP) interaction model of traditional DBMSs. Instead, its goal is to notify its users "immediately" upon the detection of a pattern of interest. Data flows are seen as streams of events, some of which may be irrelevant for the user's purposes. Therefore, the main focus is on the efficient filtering out of irrelevant data and processing of the relevant. Obviously, for such systems to be acceptable, they have to satisfy certain efficiency and accuracy constraints, such as low latency and robustness.

Numerous CEP systems have already been implemented, with very different approaches to event processing. In this paper we present an initial comparison of the widely used Esper system [1], which relies on a SQL-based language and Java, and the Event Calculus for Run-Time reasoning (RTEC) [2], a logic programming language for representing and reasoning about events and their effects. Both engines consume as their input a number of streams of low-level events, i.e. time-stamped, simple, derived events (SDE) which are themselves the product of previous computational stages on even more basic events, such as those coming from sensors. Based on these SDEs and their event representation language, the user can define the complex events (CE) of interest.

Our intention is not to build a full-scale and general benchmark for CEP systems. As of this time and to the best of our knowledge, there are no such standard benchmarks, although work towards this direction has recently appeared [10], [11]. We are rather focusing on gaining some insights with regards to the possible advantages and shortcomings of applying different event recognition approaches on a specific domain.

The rest of the paper is structured as follows. In Section 2 we present the main features of Esper and RTEC. Section 3 first describes a task for which RTEC has already been tested and then it illustrates how these systems express a class of event patterns for that task. In Section 4, the results from the comparison tests, in terms of similarity and performance, are presented and explained. Finally, in Section 5 we draw some conclusions from our tests and discuss some future work directions.

2 Complex Event Processing Engines

In this section, we briefly present the CEP engines that we investigate.

2.1 Esper

Among the currently available and well-known CEP engines, we have opted for Esper [1] (see [4], [9] and [13] for some application domains in which Esper has been used), since it is free, open-source and has already been the target of previous benchmark studies [10]. Esper is integrated into the Java and .NET languages and can be used in CEP applications as a library. For ease of understanding, one could conceptualize the Esper engine as a database turned upside-down. Traditional database systems work by storing incoming data in disks, according to a predefined relational schema. They can hold an exact history of previous insertions and updates are usually rare events. User queries are not known beforehand and there are no strict constraints as far as their latency is concerned. The Esper engine, on the other hand, lets users define from the very start the queries they are interested in, which act as filters for the streams of incoming data. Events satisfying the filtering criteria are detected in "real-time" and may be pushed further down the chain of filters for additional processing or published to their respective listeners/subscribers.

Esper provides a rich set of constructs by which events and event patterns can be expressed. One way to achieve event representation and handling is through

the use of expression-based pattern matching. Patterns incorporate several operators, some of which may be time-based, and are applied to sequences of events. A new event matches the pattern expression whenever is satisfies its filtering criteria. Another method to process events is through the Event Processing Language (EPL) queries which resemble in their syntax that of the well-known SQL. The most usual SQL constructs may also be used in EPL statements. However, the defined queries are not applied to tables but to views, which can be understood as basic structures for holding events, according to certain user demands, e.g. the need for grouping based on certain keys or for applying queries to events up to certain time point in the past.

2.2 RTEC

The Event Calculus for Run-Time reasoning (RTEC) is a logic-based CEP engine that has been successfully used in Big Data applications [3], [2]. Moving away from traditional database-like constructs, it has been written in Prolog, having as its main goal to capture the expressivity of the Event Calculus [8]. The Event Calculus is a logic formalism which extends the expressive power of logic systems so that they can handle events taking place in time. By allowing for the temporal representation of "actions" (or events), a set of rules may be built which make reasoning about time intervals, events and their relationships possible.

RTEC uses a number of techniques for increased performance and scalability. For real-time operation, a windowing mechanism may be used in order to capture events that arrive with a certain delay. Intermediate results from computations are stored in "cache" memory so that their recomputation is avoided and an indexing mechanism tunes the engine to those events that are deemed relevant.

3 Complex Event Definition

The city of Helsinki, Finland, is currently trying to develop a recognition system to support city transport management. Each vehicle of the transport network is equipped with sensors that send measurements such as arrival and departure time from a stop, acceleration information, in-vehicle temperature and noise levels. Information extracted from the sensors constitutes the SDE streams for a CEP engine. Based on these SDEs, CEs are recognized related to the punctuality of a vehicle, passenger and driver comfort, passenger and driver safety and so on. RTEC has already been tested in the domain of city transport management. More details may be found in [2].

In order to compare Esper with RTEC, we translated the main RTEC features into EPL. RTEC provides its users with four basic constructs which allow them to define the required rules for their domain: simple fluents that are subject to the law of inertia, and statically determined fluents defined in terms of three interval manipulation constructs: union, intersection and complement. Our aim is to express the RTEC features into pure EPL statements, without resorting to any Java-implemented algorithms or data structures, except of course for those holding the input streams and managing the output data.

3.1 Law of inertia

First, we give an example of how a simple fluent is defined in RTEC. The term $F=V$ denotes that fluent F has value V. For a simple fluent F, $F=V$ holds at 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), which implements the *law of inertia*. The occurrence of an event E at time T is modeled by the predicate happensAt(E,T). Interval-based semantics are obtained with the predicate holdsFor$(F=V,I)$, where I is a list of maximal intervals for which fluent F has value V continuously.

Based on the above predicates and the instantaneous SDEs (*enter_stop* and *leave_stop*) about arrival and departure times from a stop, the user can define the simple fluents for public transport vehicle punctuality with the following rules:

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

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

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

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

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

where Id is the id of a vehicle, *VehicleType* may be a bus or a tram, $Stop$ is the code of a stop, and '_' is an 'anonymous' Prolog variable.

All vehicles are initialized as being punctual. As new SDEs arrive, a vehicle becomes non punctual if it arrives late at a stop or leaves early. It becomes punctual again if it arrives early or on time at a stop. The maximal intervals for which a vehicle is considered continuously (non-) punctual are computed using the built-in/domain-independent RTEC predicate holdsFor from rules (1)–(5).

Initialization of the punctuality fluent (rule (1) in RTEC) is performed in Esper with a special *InitEvent* carrying the appropriate initial value. Rules (2)–(3) can be expressed in EPL with the following statement:

```
insert into SFEvent
select se.vehicleId as vehicleId,
       0 as sfId,
       createHash(vehicleId, 0) as hash,
       se.timestamp as timestamp,
       punctual as sfValue
from StopEvent as se
where se.eventType = eventbean.StopEventType.ENTER and
        (se.punct = eventbean.Punctuality.EARLY or
        se.punct = eventbean.Punctuality.SCHEDULED)
```
(6)

The above statement "listens" to all the SDEs related to arrivals/departures to/from stops (*StopEvent*) and keeps only the arrival events ($se.eventType = eventbean.StopEventType.ENTER$) in which the vehicle is early or scheduled. When such an event is detected, it essentially notifies the system that the vehicle has become (or remains) punctual (as in rules (2) and (3)). After this initial filtering, the statement forwards the remaining events towards the simple fluent stream (*SFEvent* in statements (6)–(7)). Each *SFEvent* is also accompanied by an attribute called *sfValue* which indicates the value of the detected event.

The statement for non punctuality, expressed in RTEC by rules (4)–(5), is written in EPL as:

```
insert into SFEvent
select se.vehicleId as vehicleId,
       0 as sfId,
       createHash(vehicleId, 0) as hash,
       se.timestamp as timestamp,
       non_punctual as sfValue
from StopEvent as se
where (se.eventType = eventbean.StopEventType.ENTER and
         se.punct = eventbean.Punctuality.LATE)
         or
         (se.eventType = eventbean.StopEventType.LEAVE and
         se.punct = eventbean.Punctuality.EARLY)
```
(7)

Due to space limitations we do not present here our domain-independent EPL code for computing the maximal intervals of simple fluents, given their starting and ending points. We briefly note that the computation method is relatively simple. A memory is maintained, holding the previously computed intervals as tuples in the form of [*startstamp*, *endstamp*, *sfValue*] and a current interval in the form of [*startstamp*, *−1*, *sfValue*], where -1 indicates a still open interval. Upon the arrival of a new interval, its *sfValue* is compared against the *sfValue* of the current interval. If they are equal, we ignore the new interval. If they are different, the current interval is closed, with the *timestamp* of the new interval replacing -1. Afterwards, a new open interval is created, beginning with the *timestamp* and the *sfValue* of the last interval. When a new interval arrives delayed, certain extra checks have to be performed as well, which we omit here.

3.2 Interval Manipulation

We now turn our attention to the interval manipulation constructs of RTEC: union, intersection and complement. Among the patterns that we would like to detect within the context of city transport management is the case when a certain vehicle is being driven in a style that is deemed unsafe. The vehicle sensors feed the system with three relevant data streams, one that informs us about the intervals during which a vehicle takes a (very) sharp turn and two more for the intervals of (very) abrupt acceleration and deceleration. The city transport management domain experts define a driving style as unsafe when a

vehicle takes a very sharp turn or is in a very abrupt acceleration or deceleration. The RTEC rule for this definition can therefore be written as follows:

$$\begin{aligned}&\mathsf{holdsFor}(driving\_style(Id, VehicleType) = unsafe,\ UDI) :- \\ &\quad \mathsf{holdsFor}(sharp\_turn(Id, VehicleType) = very\_sharp, VSTI), \\ &\quad \mathsf{holdsFor}(abrupt\_acceleration(Id, VehicleType) = very\_abrupt, VAAI), \\ &\quad \mathsf{holdsFor}(abrupt\_deceleration(Id, VehicleType) = very\_abrupt, VADI), \\ &\quad \mathsf{union\_all}([VSTI, VAAI, VADI], UDI)\end{aligned} \tag{8}$$

where Id is the vehicle identifier, $VSTI$ is the list of intervals for a very sharp turn, $VAAI$ and $VADI$ the lists of very abrupt acceleration and deceleration respectively and finally UDI is the list of intervals for unsafe driving, to be computed as the union of $VSTI$, $VAAI$ and $VADI$. The holdsFor predicate can be used in order to define the required domain-dependent rules. I in $\mathsf{union\_all}(L, I)$ is a list of maximal intervals that includes each time-point that is part of at least one list of L. Effectively, union is an implementation of OR over intervals. *sharp_turn*, *abrupt_acceleration* and *abrupt_deceleration* are streams of incoming SDEs.

Using the library that we developed for expressing the main RTEC features, rule (8) can be written:

$$\begin{aligned}&\mathsf{insert\ into}\ UnionEvent \\ &\mathsf{select}\ ste.vehicleId\ \mathsf{as}\ vehicleId, \\ &\qquad 1\ \mathsf{as}\ unionId, \\ &\qquad createHash(ste.vehicleId, 1)\ \mathsf{as}\ hash, \\ &\qquad ste.startstamp\ \mathsf{as}\ startstamp, \\ &\qquad ste.endstamp\ \mathsf{as}\ endstamp \\ &\mathsf{from}\ SharpTurnEvent\ \mathsf{as}\ ste \\ &\mathsf{where}\ ste.sharpness = eventbean.Sharpness.VERY\_SHARP\end{aligned} \tag{9}$$

The above EPL statement consumes events from the stream of *sharp_turn* SDEs, keeps only those denoting a very sharp turn and feeds the resulting output into the stream of Union events. Of course, two similar statements should also be written for the *abrupt_acceleration* and *abrupt_deceleration* streams of SDEs. These three statements together would complete the definition for unsafe driving.

Until now, we have shown how domain-dependent RTEC rules can be expressed as domain-dependent EPL statements. It is worth commenting that RTEC rules can be expressed in a purely declarative and more compact way than EPL statements. Additionally, when writing EPL statements, the user needs to have at least some elementary knowledge of how events are represented.

3.3 EPL library

In order to better understand the functionality of the *unionId* and *hash* attributes, we take a closer look at the internals of our application-independent EPL library. EPL can use such SQL-like statements to filter event streams and/or to push the results of filtering further down to other streams. Besides this basic functionality, EPL can also make use of the so-called views, which are similar

to SQL tables and can hold multiple events. More complex operations can be performed on these views, such as aggregation and grouping. For our present purposes, we are interested in time-based views whose expiry policies (when to remove an event) employ time windows.

The RTEC constructs (inertia, interval union, intersection and relative complement) that were translated to EPL follow the same pattern for computing the maximal intervals of a fluent. For each RTEC construct, we maintain a memory holding the previously computed disjoint intervals up to certain time point in the past. Under the assumption that the incoming event intervals arrive in an orderly manner, such a memory would be redundant, since we would need to maintain only a single interval and update it, in case a new event overlaps with it or simply release it as a final result if there is no overlap. However, there are events which arrive delayed, affecting in this way the previously computed intervals. In order to be able to handle delayed events, we need to store the intermediate results, at least up to a certain time point. RTEC has a sliding window approach to deal with such delayed information. For the same reason, we need to use the time-based views provided by EPL. Whenever a new event arrives (for our union example, a *UnionEvent*, similarly for the other three constructs), we first check whether and how it affects any of the previously computed intervals. Any new insertions, deletions or updates are performed according to the results of this checking step.

An additional memory/view for storing intervals of previous events as well, besides the computed intervals, is required for the operations of intersection and complement. In these cases, the interval of a new event may not only interact with the previously computed intervals but with previous events too. Consider, for example, intersection. I in $\mathsf{intersect\_all}(L, I)$ of RTEC is a list of maximal intervals that includes each time-point that is part of all lists of L. In Esper, a new *IntersectionEvent* may not overlap at all with any of the stored intervals but we still cannot deduce that it should be ignored. An overlapping event may have appeared previously which, at the time of its appearance, had no effect and was not involved in the construction of an interval. On the other hand, the union operation is additive and the unionized intervals implicitly take into account all previous events. All time-points included in previous *UnionEvents* (in their intervals of $[startstamp, endstamp)$) are also to be found in the stored intervals and there is no need for additional memories.

For the union operation, the definition of its time-based view is shown in statement (10). This statement simply creates a view (*window*), based on the attributes of the *UnionEvent*. Its purpose is to store the currently computed intervals for the union operation. According to this statement alone, it has no expiry policy ($.win : keepall()$) and keeps all intervals until they are explicitly deleted.

This view is initially empty. Upon the arrival of a new *UnionEvent*, a merge operation is performed which is the equivalent of the SQL upsert (update and insert) operation. EPL statement (11) performs the merge operation.

The *where* clause of statement (11) compares all the stored intervals of the *UnionWindow* with the new *UnionEvent* in order to determine whether any of the intervals overlap with the new event. If there is no such interval then the new event may simply be inserted as a new interval. If there are affected intervals, then a new event (*CheckAffected*) is created whose purpose is to collect information about those intervals. We omit this step here to save space. We just note that out of the affected intervals (including the newly arrived interval), what we need to know is their minimum *startstamp* and their maximum *endstamp*. Every interval that falls between these two time points is subsequently deleted (we omit the presentation of the deletion statement).

Finally the unionized interval, simply defined as [*minstartstamp*, *maxendstamp*), is stored, according to statement (12) (the *AffectedIntervals* event holds the required information and is triggered by the *CheckAffected* event).

A similar procedure is followed for the other three constructs as well, with a slight difference. In the cases of intersection and complement, besides storing intervals, we also need to store previous events.

```
context UnionContext
create window UnionWindow.win:keepall()
select vehicleId as vehicleId, unionId as unionId, hash as hash,
    startstamp as startstamp, endstamp as endstamp
from UnionEvent
```
(10)

```
context UnionContext
on UnionEvent ue
merge UnionWindow uw
where (ue.startstamp >= uw.startstamp and ue.startstamp <= uw.endstamp
      and ue.endstamp >= uw.endstamp) or
      (uw.startstamp >= ue.startstamp and uw.endstamp <= ue.endstamp) or
      (ue.endstamp >= uw.startstamp and ue.endstamp <= uw.endstamp
      and ue.startstamp <= uw.startstamp) or
      (ue.startstamp >= uw.startstamp and ue.endstamp <= uw.endstamp)
when not matched
then insert select ue.vehicleId as vehicleId, ue.unionId as unionId,
                   ue.hash as hash,
                   ue.startstamp as startstamp, ue.endstamp as endstamp
when matched
then insert into CheckAffected select ue.vehicleId as vehicleId,
                                      ue.unionId as unionId,
                                      ue.hash as hash,
                                      ue.startstamp as startstamp,
                                      ue.endstamp as endstamp
```
(11)

context $UnionContext$
on $AffectedIntervals\ ai$
merge $UnionWindow\ uw$
where $uw.startstamp = ai.minstartstamp$ and $uw.endstamp = ai.maxendstamp$
when not matched
then insert select $ai.vehicleId$ as $vehicleId$, $ai.unionId$ as $unionId$,
$ai.hash$ as $hash$, $ai.minstartstamp$ as $startstamp$,
$ai.maxendstamp$ as $endstamp$

(12)

Statements 10–12 make use of contexts, as seen at their first lines. Contexts are an EPL concept for partitioning windows according to a specified key (we have omitted the declaration of the *UnionContext* here). In our case, we have used the hash attribute, created upon the *unionId* and *vehicleId*, as the partitioning key. This has the effect that each unique combination of the *unionId* and *vehicleId* attributes has a separate *UnionWindow* (essentially a separate memory for each combination), although we only need to define it once. Whenever a new event is pushed into the *UnionEvent* stream, as in statement (9), the Esper engine makes a choice as to which *UnionWindow* it should be sent, according to its *hash* value. For example, *UnionEvents* with a value of *vehicleId* equal to 75 and participating in the union operation with identifier equal to 1, giving us a hash value of 18880, are fed only to the *UnionWindow* with the same hash value. If the *UnionWindow* was one and the same for all vehicles and unions, then statement (11) would refer to this single window, holding all union intervals. Its where clause would have to include an extra condition, ($ue.vehicleId = uw.vehicleId$ and $ue.unionId = uw.unionId$) so that the new event is checked only against the intervals related to this specific *vehicleId* and *unionId*. By creating different windows for each value of the partitioning key, contexts implement a more efficient indexing mechanism than checking in a single window for the right combination of *vehicleId* - *unionId*. Additionally, contexts are more amenable to parallelization when multiple threads are available.

Of course, for real-time operation, an "infinite" memory that never deletes intervals and/or events would be impractical. To address this issue, we introduced a windowing parameter, called working memory, as in RTEC, and two special events, the *QueryEvent* and the *ClearEvent*. At each query step (e.g. defined as 1000 time steps), a *QueryEvent* is first sent to the Esper engine, followed by a *ClearEvent*. The *QueryEvent* releases all the intervals which fall outside of the working memory window as final results and the *ClearEvent* deletes those intervals.

3.4 Event Hierarchy Representation

Figure 1 depicts the event hierarchy of the city transport management application, that is, the SDE streams (denoted by incoming arrows on the left), the

operations in which they are involved (denoted by boxes) and the CEs to be detected (denoted by outgoing arrows from the operation boxes). The results from a certain operation can be fed into another operation, so that complex chains of rules may be implemented.

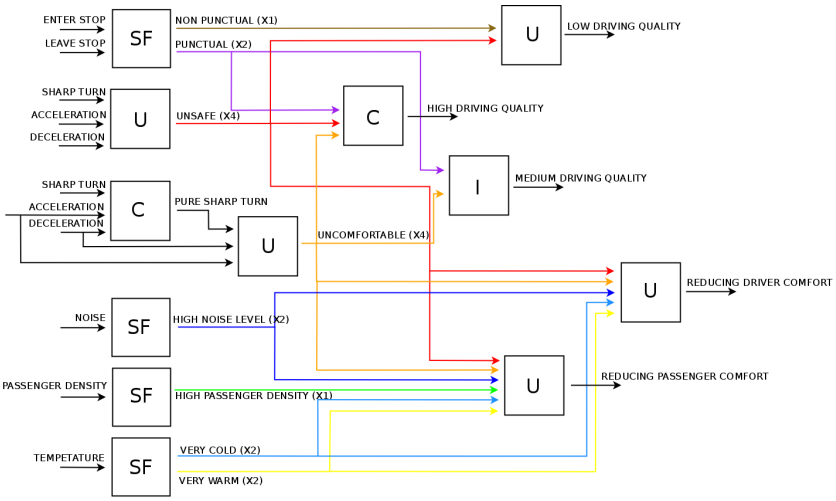


Fig. 1. The event hierarchy of the city transport management application. SF=simple fluent, U=union, I=intersection, C=complement. The number besides each CE stream indicates its fanout.

An important difference between RTEC and Esper in their functioning is the way in which they process information from new events. RTEC does not perform any computations before a query is "triggered". Being written in Prolog, it essentially employs a pull method for retrieving information from events. At the time of a query, it backtracks in order to satisfy its goals which describe the CEs of interest. On the other hand, Esper employs a push method, performing on-the-fly processing of new events. Even if a *QueryEvent* has not appeared yet, new SDEs are directly pushed towards their respective windows and the *QueryEvent* triggers the release of the computed intervals. However, there are some exceptions to on-the-fly processing, whenever there are event hierarchies in which an operation requires the results of a previous operation. In these cases, an operation must wait for the final results of the previous one. For example, in Figure 1, the *uncomfortable* CE, depends both on two of the SDE streams and on the intermediate *pure_sharp_turn* stream. Before a *QueryEvent* arrives, the operation for *uncomfortable* can unionize the two SDE streams but it has to wait for the *QueryEvent* in order to include the *pure_sharp_turn* stream.

4 Empirical Evaluation

For the tests that follow, we used three different synthetic datasets of SDEs, based on the city transport management task described above and generated

during the PRONTO project (`http://www.ict-pronto.org`). Each dataset consists of 50.000 SDEs. In the first dataset, all SDEs refer to a single vehicle, the second contains events for 10 vehicles and the third for 100. In total, we have 8 incoming SDE streams and 13 CE streams.

4.1 Similarity Testing

In order to assess the similarity of the results produced by RTEC with those produced by Esper, we ran a series of tests on the City Transport Manaagement datasets. Using the intervals computed by RTEC as a reference point, we compared them with the intervals produced by Esper. The comparison metric for each CE is computed as the division of the intersection of the RTEC and Esper intervals by their union. Figure 2 presents the comparison results for the three different datasets.

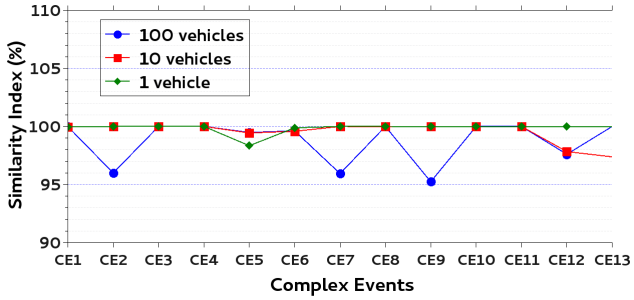


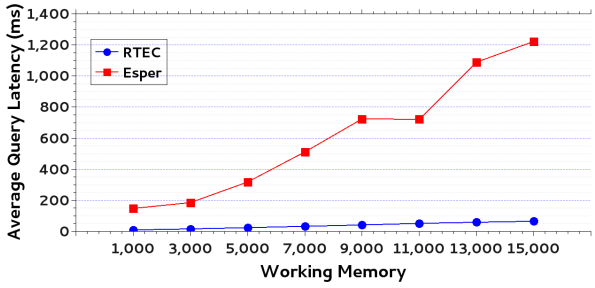
Fig. 2. Esper/RTEC similarity index for three different datasets.

For most of the CEs in all datasets, the similarity is near perfect (above 99%), while for all of them it lies above 95%. For a few of them, especially in the dataset with 100 vehicles, it falls to a level of about 95%. These discrepancies are due to a slight difference between RTEC and Esper in the way open intervals are treated, i.e. intervals which, at the time of a query, have not been closed, e.g. when a sharp turn has not finished when a query is triggered. In this case, RTEC produces intervals in the form of $[startstamp, inf)$ which are ignored during the comparison, whereas Esper produces intervals in the form of $[startstamp, querytime)$. In the dataset with 100 vehicles, due to its structure, such open intervals appear more often.

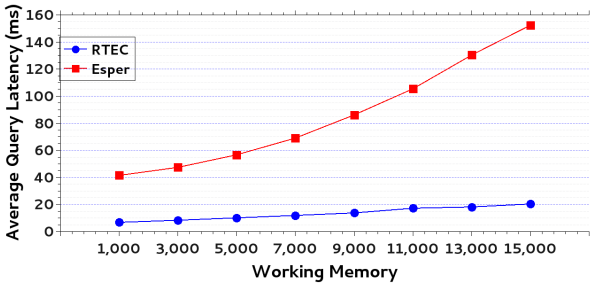
4.2 Performance Comparison

After ensuring that Esper can reproduce the RTEC results with an accepted level of reliability, we ran another series of experiments in order to compare their performance, in terms of average latency per query. All the experiments were conducted on a machine with an Intel Core 2 Duo CPU P8600 @ 2.40 Ghz x 2 processor and 3.0 Gib of memory, running the 32-bit version of Debian 7.3. Esper was run with java-7-openjdk-i386 and RTEC with YAP Prolog 6.2.2. Both engines were tested as single-threaded applications. Figures 3(a)–3(c) depict the results of the performance comparison tests, with each figure referring

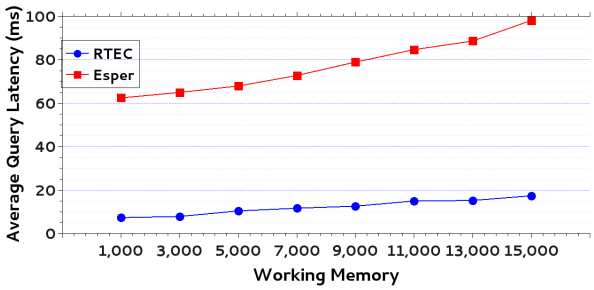
to a different dataset. In order to assess the impact of the size of the working memory window (how far into the past an operation can look into when computing intervals), we varied its value from 1000 (equal to the query step) to 15000 time-points.



(a) 1 vehicle



(b) 10 vehicles



(c) 100 vehicles

Fig. 3. Esper/RTEC performance comparison

Figures 3(a)–3(c) show that RTEC outperforms Esper significantly, with Esper suffering a worst degradation as the working memory window increases (increased gradient). By separating the time of on-the-fly processing from that of query-time processing (see Section 3 for an explanation of their difference), we discovered that the observed latency is almost exclusively due to query-time processing. In an attempt to isolate the possible bottlenecks, we measured latencies

on a per operation basis. The results did not indicate that a specific type of operation is a significant source of delay. We observed that the latency of an operation correlates with its fanout. The operations for the *unsafe* and the *uncomfortable* CEs, both with a fanout of 4 (see Figure 1), were consistently among the most severe bottlenecks. Therefore, we are inclined to assume that one reason behind Esper's lower performance lies in the "communication" overhead between the connected operations.

5 Conclusions

We presented a comparison of two CEP engines with significant differences as far as their event representation languages are concerned. As CEP engines mature over time, such comparison tests are expected to become more common. They will allow for the identification of the possible limitations and advantages of one solution over another and will facilitate the classification of event-based systems [12].

For the CEP engines under comparison here —Esper and RTEC—, we showed that the translation of an event hierarchy from one language to the other, although not a trivial task, is certainly possible. In fact, for certain domains, such as city transport management above, the whole process could be delegated to automatic translators. However, when the task of translation is left to the user, EPL statements are longer and require some low-level knowledge of how events are represented, which could make them more susceptible to subtle errors. Contrary to what one might expect, we also showed that a system from the field of Artificial Intelligence, like RTEC, may outperform a state-of-the-art system from the fields of databases and distributed systems.

Our tests are far from being complete and we do not consider the results presented here to be final. RTEC is very well-suited for problems whose main goal is to find those time intervals during which certain conditions hold. Moreover, translating the RTEC constructs directly into EPL statements might very well result in an inefficient implementation from the point of view of Esper. However, this is exactly one of our goals, i.e. to find those domains for which a certain CEP system might be more appropriate than others.

For these reasons, we would like to continue this line of work. We aim to investigate, for example, whether the recently introduced data flow programming model of Esper improves performance for the type of event definitions that were examined in this paper. Additionally, we will compare the two engines using event patterns that are more readily expressed in EPL. We will also compare the two systems, along multiple dimensions, both in terms of semantics and performance (see [14], [10], [7] for examples of studies with a more detailed treatment of these issues).

Acknowledgments

We have benefited from discussions with Matthias Weidlich on the comparison of Esper and RTEC. This work has been funded by the EU SPEEDD project (FP7-ICT 619435).

References

1. Esper reference document. `http://esper.codehaus.org/esper-4.10.0/doc/reference/en-US/html/index.html`. Accessed: 2014-01-21.
2. A. Artikis, M. J. Sergot, and G. Paliouras. Run-time composite event recognition. In *DEBS*, pages 69–80, 2012.
3. A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-adaptive event recognition for intelligent transport management. In *2013 IEEE International Conference on Big Data*, pages 319–325, 2013.
4. B. Balis, B. Kowalewski, and M. Bubak. Real-time grid monitoring based on complex event processing. *Future Generation Computer Systems*, 27(8):1103–1112, Oct. 2011.
5. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.
6. O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Company, 2010.
7. T. Grabs and M. Lu. Measuring performance of complex event processing systems. In *Topics in Performance Evaluation, Measurement and Characterization*, page 8396. Springer, 2012.
8. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
9. T. Ku, Y. Zhu, and K. Hu. Semantics-based complex event processing for RFID data streams. In *The First International Symposium on Data, Privacy, and E-Commerce, 2007. ISDPE 2007*, pages 32–34, 2007.
10. M. R. Mendes, P. Bizarro, and P. Marques. A performance study of event processing systems. In *Performance Evaluation and Benchmarking*, page 221236. Springer, 2009.
11. M. R. Mendes, P. Bizarro, and P. Marques. Towards a standard event processing benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page 307310, New York, NY, USA, 2013. ACM.
12. A. Voisard and H. Ziekow. Architect: A layered framework for classifying technologies of event-based systems. *Inf. Syst.*, 36(6):937–957, 2011.
13. S. Weber, H. J. Lowe, S. Malunjkar, and J. Quinn. Implementing a real-time complex event stream processing system to help identify potential participants in clinical and translational research studies. *AMIA Annu Symp Proc*, 2010:472–476, 2010. PMID: 21347023 PMCID: PMC3041381.
14. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 407418. ACM, 2006.