GeoInformatica (2026) 30:9
https://doi.org/10.1007/s10707-026-00569-z

RESEARCH

Online spatial reasoning for complex event recognition

Elias Alevizos[1,4] · Georgios M. Santipantakis[2] · Christos Doulkeridis[2] · Alexander Artikis[3,4]

Received: 30 September 2024 / Revised: 22 May 2025 / Accepted: 17 February 2026

Abstract
Complex Event Recognition (CER) systems have the ability to process streams of events by detecting event patterns with minimal latency. Typically, these patterns have a temporal structure, often resembling the sequential structure of regular expressions. A pattern advances to the next state by checking various conditions on the current and possibly previous events of the stream. CER systems are very efficient in tracking all the possible paths that a pattern may follow and report when a path is complete and a complex event must be reported. In some cases, the conditions that need to be checked may be spatial. For example, in maritime situational awareness, a condition may need to check whether a vessel is close to any other vessel. Such conditions are not easily expressed directly as regular expressions. For such spatio-temporal tasks, there exist dedicated modules which can evaluate this type of conditions efficiently. Thus, we can integrate such a spatio-temporal module within a CER system in order to take advantage of both worlds: the CER engine can accommodate and process complex regular expressions and delegate the evaluation of expensive spatio-temporal tasks to a dedicated module whenever it needs to. We present an approach towards such an integration. We describe how a CER engine, based on symbolic automata, can cooperate with a spatio-temporal link discovery (stLD) module such that the former can leverage the spatio-temporal capabilities of the latter. This cooperation can take place in an online manner rendering the whole system suitable for real-time processing of event streams. We discuss two different communication schemes between the CER engine and the spatio-temporal module and explore when each one should be preferred. We provide a theoretical estimation of the predicted performance of the system under each communication scheme. Our extensive experimental evaluation confirms most of our theoretical predictions.

Keywords Finite automata · Complex event recognition · Complex event processing · Spatial reasoning · Geospatial interlinking · Spatiotemporal link discovery

Extended author information available on the last page of the article

Published online: 03 March 2026

1 Introduction

Complex Event Recognition (CER) systems consume streams of simple, input events in real time and produce another stream of output, complex events [1–3]. The detection of these complex events is driven by a set of patterns, defining relations among the input events. The patterns determine how the input events must be ordered in time for them to be considered a pattern match. Patterns are typically defined through a temporal formalism. For example, variations of regular expressions and automata are often employed to express complex event patterns. Besides temporal relations, a pattern may also define non-temporal constraints on the input events, e.g., that two moving objects are close in space. The input to a CER system thus is a) a stream of simple, input events; b) a set of patterns that define relations among the input events. Instances of pattern satisfaction are called complex events. The output of the system is another stream, composed of the detected complex events. One main requirement for CER systems is that they must detect complex events with very low latency, which, in certain cases, may even be in the order of a few milliseconds [2, 4, 5].

As an example, consider the case of maritime situational awareness, which has gained considerable attention in the past years, both for economic and for environmental reasons [6–10]. Monitoring vessel activity is critical for preventing risks and detecting suspicious or illegal behaviours, due to the Automatic Identification System (AIS) [11]. AIS is used to track vessels at sea in real-time by allowing vessels to emit information about their status (e.g., position and velocity) to other vessels as well as to coastal stations. The system that we present in this paper focuses on the domain of maritime monitoring and its purpose is to inform potential analysts about the behaviour of vessels at sea.

In this context of real-time monitoring of moving vessels, spatial reasoning is also important as it enables the computation of complex relations between spatial entities. For instance, the system needs to discover vessels that come close to each other (possibly indicating a collision risk) or vessels that sail very close to the coastline, to issue an alert. For this purpose, in previous work, we have developed a prototype for real-time spatial and spatio-temporal reasoning, called stLD [12], which can compute topological or proximity relations between various types of spatial entities (points, lines, polygons) with low latency. Moreover, stLD supports link discovery operations, meaning that it can consume semantic representations of data (using RDF), perform spatial reasoning and provide its output in the form of interlinked data.

As described above, the stLD component performs semantic spatio-temporal reasoning on raw stream data. On the other hand, a CER system performs real-time reasoning at a higher abstraction level in order to detect interesting activity patterns. Our intention is to investigate how a CER system could establish communication with a stLD engine so that it can take advantage of such an engine's reasoning capabilities. Thus, the CER system will be able to focus more on performing high-level temporal reasoning, whereas the stLD can provide support for lower-level spatio-temporal reasoning. The result will be a high performance unified engine that can perform reasoning at various levels by taking into account and integrating in real time any available background knowledge.

This paper is an extension of the work presented in [13]. In [13], we presented an initial version of our system. We described at a high level of abstraction the CER and stLD modules and their interactions, without providing details about their internal workings. Our experimental evaluation was also relatively limited, focusing on a single pattern. This paper

extends on our previous work. On the one hand, we clarify the operation of stLD by presenting its internals and the type of supported relations and we present the details about the interaction between CER and stLD, as well as an analysis of the performance of the overall system. On the other hand, we have extended the experimental evaluation, by including experiments for stLD about proximity between trajectories and for the overall CER system using extended patterns that need to be detected. Additionally, we present a theoretical analysis regarding the applicability of the various communication schemes between CER and stLD. We empirically demonstrate in which cases and to what extent our analysis is confirmed by the experiments.

The paper is structured as follows: Section 2 briefly presents previous related work on spatio-temporal link discovery and on CER enriched with spatio-temporal capabilities. In Section 3 we give an overview of the CER and stLD modules and describe how they function in isolation. We then present how these two modules can cooperate and communicate with each other in Section 4. In Section 5 we present experimental results and we conclude with Section 6.

2 Related work

In this section, we discuss previous work regarding link discovery, complex event recognition and their possible combinations and integration approaches.

2.1 Link discovery

Spatial link discovery or geospatial interlinking has been studied in [14–16]. The objective is to discover relations of spatial nature between two datasets in an efficient way. Typically, most approaches in the literature rely on *blocking* techniques that organize data in memory (e.g., using space tiling), so as to quickly perform a *filtering step* that produces a small number of candidate pairs. Then, in the *refinement step*, these candidates are examined by evaluating the spatial relation, in order to return the true results. More recent approaches focus on progressive interlinking to discover as many links as possible for a given budget [17, 18]. In the same line of work, geospatial interlinking assisted by supervised learning has been studied in [19]. Notably, there is much less work for spatio-temporal link discovery [12, 20] which involves the movement of objects, and for real-time link discovery where the links need to be discovered over streaming input sources.

2.2 Complex event recognition

Complex Event Recognition systems have been studied extensively in the past decades (see [1, 3] for reviews) and they come in various colours and flavours. The overarching theme is the requirement for real-time detection of complex temporal patterns on high velocity and high volume streams of events. A significant number of CER systems employ automata as their computational model, whereas some other resort to logic-based solutions or tree structures. A feature that is generally lacking though is the ability of a CER engine to efficiently take into account any background knowledge [1], e.g., bathymetry data in the maritime domain so as to avoid possible grounding incidents. This is not knowledge that is

directly present in the payload of the input events, but may be static knowledge residing in a (possibly remote) database. Moreover, there is the need to combine this static knowledge with dynamic information, e.g., (static) bathymetry data with (dynamic) position signals to determine whether a vessel is in danger. In addition, we may also need to extract complex spatio-temporal information, not necessarily related to any static datasets, e.g., to check whether a vessel is in close proximity to any other vessel at a given point or interval in time.

2.3 Complex event recognition with spatio-temporal awareness

One solution to these issues is to initially pre-process the stream of input events with a dedicated module which can perform this type of knowledge integration. We can then generate a new stream, enriched with background knowledge. This enriched stream can then be used to perform CER [10, 21, 22]. However, it is not clear whether such an approach could work efficiently in a proper streaming environment, with events arriving in the system in real time. Another approach is to have any background knowledge reside in remote databases and access the required information on a need-to-know basis, as in [23]. While this is a method which can indeed work in real time, its limitation is that it can work only with static knowledge, i.e., we can extract remote information and use it in our pattern constraints, but we cannot perform any reasoning on that information before it reaches the CER system.

In [24], a CEP system is proposed, called GeoT-Rex, that is endowed with spatial capabilities. The systems can define Geometry as a new, spatial data type. It also allows users to employ several functions dedicated to spatial data operations. The spatial capabilities are therefore tightly coupled with the main CEP engine. In contrast to this approach, we have opted for a more loosely coupled architecture. The reason is that this choice allows for greater flexibility. First, the CEP and stLD modules can work and, thus, be optimized independently. Given that stLD and CEP may have different computational demands, this allows for better load balancing. Additionally, the CEP and stLD modules may even be located in separate machines and be geographically distributed. Finally, with a loose architecture, the language of one system (e.g., the CEP engine) does not need to closely follow and reflect possible changes in the language of the other (e.g., the stLD module). The only requirement is that a new stLD predicate has an appropriate API implementation inside the CEP engine, without the need to specify new data types.

In [25], a spatio-temporally aware CEP system is proposed. CEP is enriched with complex operators that are important for the analysis of maritime data. Specifically, temporal operators are introduced for handling durative events CEP rules. Spatial operations are also included, which may define relationships between position data. The new spatio-temporal capabilities are again tightly integrated with the main CEP engine. This implies that the same limitations described above still hold. We have rather opted for a more loosely coupled architecture that offers several advantages, like the ability for geographically distributed processing, access to remote modules, optimization flexibility and language independence.

Our approach focuses on a scheme of labour division between CER and specialized spatio-temporal processing. While the CER system remains responsible for handling the general temporal structure of a pattern, it delegates expensive spatio-temporal tasks to a dedicated module whenever there is such a need. This module has the ability to provide both static information but can also perform complex spatio-temporal reasoning tasks.

3 Background

In this Section, we present the necessary background information about the CER and stLD engines that we have used. We first briefly describe the engines when they function in isolation in order to aid understanding. In Section 4, we will show how they can work in conjunction.

3.1 Complex event recognition

We begin by first presenting the CER engine we have adopted. We have chosen to use Wayeb, a Complex Event Recognition and Forecasting engine which employs symbolic automata as its computational model [26–28]. Wayeb is both efficient and expressive, while maintaining clear, compositional semantics for the patterns expressed in its language due to the fact that symbolic automata have nice closure properties [29]. At the same time, it is expressive enough to support most of the common CER operators [1].

Wayeb's patterns are expressed as symbolic regular expressions, i.e., they are regular expressions with the important difference that its terminal expressions are not simple symbols from an alphabet, but Boolean expressions [26]. Wayeb's standard operators are those of the classical regular expressions, i.e., concatenation, disjunction and Kleene-star. Wayeb's language has been extended to include various extra CER operators, e.g., that of negation and those of different selection policies (see [1] for a discussion of selection policies). Wayeb can also use windows that specify the temporal interval within which a set of events may occur so as to be considered as a valid complex event.

Formally, symbolic regular expressions are defined as follows:

Definition 1 (Symbolic regular expression) A Wayeb symbolic regular expression (SRE) is recursively defined as follows:

- If ψ is a Boolean expression, then $R := \psi$ is a symbolic regular expression, with $\mathcal{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the subset of all possible elements / simple events for which ψ evaluates to TRUE;
- Disjunction / Union: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 + R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$;
- Concatenation / Sequence: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 \cdot R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, where $\cdot$ denotes concatenation. $\mathcal{L}(R)$ is then the set of all strings constructed from concatenating each element of $\mathcal{L}(R_1)$ with each element of $\mathcal{L}(R_2)$;
- Iteration / Kleene-star: If R is a symbolic regular expression, then $R' := R^*$ is a symbolic regular expression, with $\mathcal{L}(R^*) = (\mathcal{L}(R))^*$, where $\mathcal{L}^* = \bigcup_{i \geq 0} \mathcal{L}^i$ and $\mathcal{L}^i$ is the concatenation of $\mathcal{L}$ with itself i times.
- Negation / complement: If R is a symbolic regular expression, then $R' := !R$ is a symbolic regular expression, with $\mathcal{L}(R') = (\mathcal{L}(R))^c$.
- skip-till-any-match selection policy: If $R_1, R_2, \cdots, R_n$ are symbolic regular expressions, then $R' := \#(R_1, R_2, \cdots, R_n)$ is a symbolic regular expression, with $R' := R_1 \cdot \top^* \cdot R_2 \cdot \top^* \cdots \top^* \cdot R_n$.

- skip-till-next-match selection policy: If $R_1, R_2, \cdots, R_n$ are symbolic regular expressions, then $R' := @(R_1, R_2, \cdots, R_n)$ is a symbolic regular expression, with $R' := R_1 \cdot !(\top^* \cdot R_2 \cdot \top^*) \cdot R_2 \cdots !(\top^* \cdot R_n \cdot \top^*) \cdot R_n$.

Wayeb patterns are defined as symbolic regular expressions which are subsequently compiled into symbolic automata. Symbolic automata resemble classical automata to a large extent. The main difference is that their transitions, instead of being labelled with a symbol from an alphabet, are equipped with logical formulas in the form of Boolean expressions. For a symbolic automaton to move to another state, it first applies the Boolean expressions of its current state's outgoing transitions to the element last read from the stream. If an expression evaluates to TRUE, then the corresponding transition is triggered and the automaton moves to that transition's target state. The definition for a symbolic automaton is the following:

Definition 2 (Symbolic finite automaton [29]) A symbolic finite automaton (SFA) is a tuple $M =(Q, q^s, F, \Delta)$, where Q is a finite set of states; $q^s \in Q$ is the initial state; $Q^f \subseteq Q$ is the set of final states; $\Delta \subseteq Q \times \Psi \times Q$ is a finite set of transitions. Ψ is the set of Boolean expressions that can be constructed from a set of predicates with the standard Boolean connectors, i.e., conjunction, disjunction and negation.

A sequence $w = t_1 t_2 \cdots t_k$, where t_i are simple events, is accepted by a SFA M iff, there exists a run (i.e., a sequence of transitions) $q_0 \xrightarrow{t_1} q_1 \cdots q_{i-1} \xrightarrow{t_i} q_i \cdots \xrightarrow{t_k} q_k$ such that $q_0 = q^s$ and $q_k \in Q^f$. In other words, if the SFA reaches a final state upon reading a sequence of events, then this sequence is accepted by the SFA. The set of sequences accepted by M is the language of M, denoted by $\mathcal{L}(M)$. We can now define "complex events". A stream S is an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a simple event. Our final goal is to report the indices i at which a complex event is detected. If $S_{1..k} = \cdots, t_{k-1}, t_k$ is the prefix of S up to the index k, we say that an instance of a SRE R is detected at k iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in \mathcal{L}(R)$.

Example 1 As an example, consider the domain of maritime monitoring. An analyst could use the Wayeb language to define the pattern $R := (speed > 10) \cdot (speed > 10)$ in order to detect speed violations in certain designated areas where the maximum allowed speed is $10\ knots$. This pattern detects two consecutive events where the speed exceeds the threshold in order to avoid cases where a vessel momentarily exceeds the threshold, possibly due to some measurement error. This is necessarily a simplified version of a speed violation pattern in order to demonstrate in an accessible manner the way Wayeb works. This pattern would be compiled to the (non-deterministic) automaton of Fig. 1. Table 1 shows an example stream processed by this automaton. For the first three input events, the automaton would remain in its start state, state 0. After the fourth event, it would move to state 1 and after the fifth event it would reach its final state, state 2. We would thus say that a complex event R was detected at $\mathrm{timestamp} = 5$.

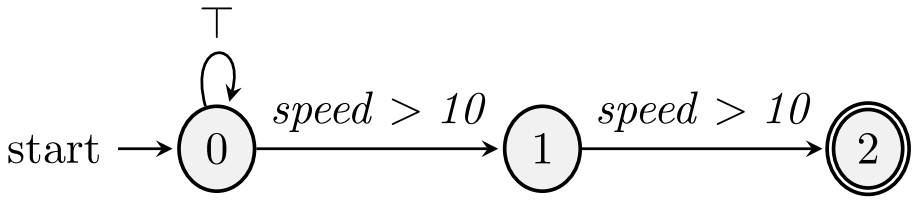


Fig. 1 Streaming symbolic automaton created from the expression $R := (speed > 10) \cdot (speed > 10)$

Table 1 An example stream composed of six events. Each event has a vessel identifier, a value for that vessel's speed and a timestamp

Vessel id	78986	78986	78986	78986	78986	...
Speed	5	3	9	14	11	...
Timestamp	1	2	3	4	5	...

3.2 Spatiotemporal link discovery

In this Section, we present preliminary concepts of spatial link discovery (Section 3.2.1), we explain the filtering (Section 3.2.2) and refinement (Section 3.2.3) phases of stLD, and we discuss how these phases are extended to support link discovery in a streaming setting (Section 3.2.4).

3.2.1 Preliminaries

Link discovery (LD) is the process of identifying relations between entities that originate from different data sources [30]. In the case of spatial or spatiotemporal data represented by geometries, which is the focus of this paper, the relations capture topological or proximity relations between geometries. Formally, given two data sources, usually referred to as *target* ($\mathcal{T}$) and *source* ($\mathcal{S}$), describing spatiotemporal entities, and a set of relations ($\mathcal{R}$), the objective of *spatiotemporal link discovery* is to compute pairs of entities $\langle s, t \rangle \in r$, where $s \in \mathcal{S}, t \in \mathcal{T}$ and $r \in \mathcal{R}$. Figure 2 illustrates two types of proximity relations used in this paper. Entities of $\mathcal{T}$ and $\mathcal{S}$ data sets are shown in different colors and pairs of entities satisfying the relations are connected with dashed lines. A brute-force approach that compares all pairs of entities for each relation has high complexity: $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{T}| \cdot |\mathcal{R}|)$, since a spatial relation is typically costly and cannot be computed in constant time. Obviously, algorithms that perform this link discovery task in an efficient and scalable way are of interest.

To address this challenge, we have proposed a framework (stLD [12, 31]) that supports a wide variety of spatial/spatiotemporal relations, both for archival data as well as for streaming data. The stLD framework relies on the filtering-and-refinement methodology to compute the relations of interest. It has been used for computing topological (Dimensionally Extended 9-Intersection Model, DE-9IM) and spatiotemporal proximity relations between any type of geometries. In this paper we focus on point-to-point and segment-to-segment proximity relations, using Haversine distance and user-defined spatial and temporal thresholds.

In the *filtering phase*, space tiling is used to partition the 2D data space in tiles, aiming at checking the existence of relations only among entities that belong to each tile. This pro-

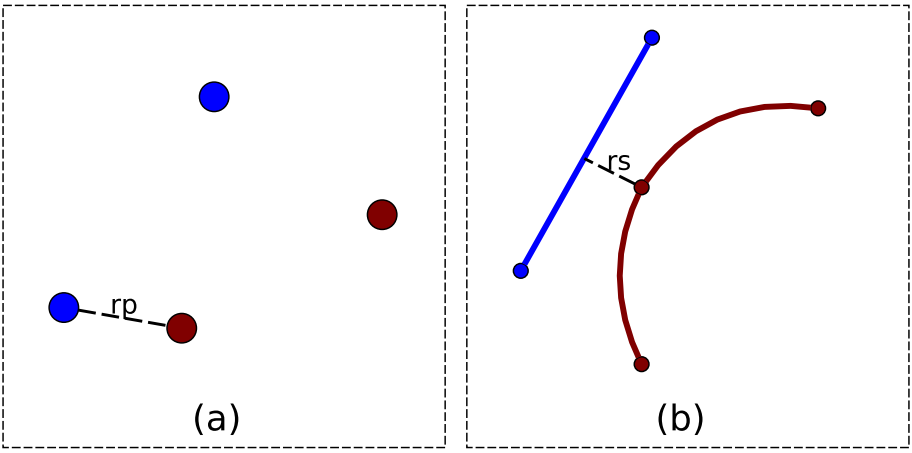


Fig. 2 Illustration of spatiotemporal (**a**) point-to-point and (**b**) segment-to-segment proximity relations discussed in this work, colors indicate different entities

duces a set of *candidate pairs* that are passed over to the *refinement phase*, which is responsible for evaluating the complex spatial relation for each candidate pair. The candidate pairs for which the spatial relation holds form the final result set, whereas the remaining pairs are discarded.

3.2.2 Filtering phase using space-tiling

The stLD implements an equi-grid for space tiling, which can be applied on any type of geometry (point, polyline, polygon). Consider the 2D data space $[x_L, x_U] \times [y_L, y_U]$ that contains the two input data sets. An equi-grid is constructed by partitioning the X-axis (Y-axis) in m_x (m_y) equi-sized splits, thus forming $m_x \cdot m_y$ tiles.

In the case of a spatial point with coordinates (x,y), we can derive the enclosing tile $c(i,j)$ for any given spatial point (x,y) in constant time as follows:

$$i = \left\lfloor \frac{x - x_L}{\delta x} \right\rfloor, j = \left\lfloor \frac{y - y_L}{\delta y} \right\rfloor$$

where $\delta x = \frac{x_U - x_L}{m_x}$ and $\delta y = \frac{y_U - y_L}{m_y}$. Let $b(x, y)$ denote a function that computes the enclosing tile $c(i, j)$ for a given point with coordinates (x, y), i.e., $b(x, y) = c(i, j)$.

In the case of a polyline (sequence of positions) $g = \{(x_0, y_0), ..., (x_k, y_k)\}$ of an entity, the set of enclosing tiles $C(g)$ can be computed by applying the function $b(x, y)$ for each point and (if necessary, for some of) the interpolated points between successive positions.[1] In the case of a polygon g, representing a region of interest, which is also defined by a set of points, the set of enclosing tiles $C(g)$ can be computed from the minimum bounding rectangle (MBR) $m(g)$ of its geometry $m(g) = \langle (mx_L, my_L), (mx_U, my_U) \rangle$, as follows:

[1] This is necessary in case two successive points of g belong to two different tiles, and there exist other tiles that intersect with the line formed by the two points.

$$C(g) = \{c(i,j) | i \in [i_{min}, i_{max}], j \in [j_{min}, j_{max}]\}, \text{where}$$
$$c(i_{min}, j_{min}) = b(mx_L, my_L) \text{ and } c(i_{max}, j_{max}) = b(mx_U, my_U)$$

It follows that given any geometry g, we can always determine the non-empty set of tiles $C(g)$, such that each $c \in C(g)$ either encloses the geometry g or overlaps with any part of g. We say that the geometry g is *assigned* to tiles $C(g)$, and that the tiles $C(g)$ *contain* the geometry g.

Example 2 Figure 3 illustrates two trajectories of vessels using different colors. The two polylines are constructed based on the individual positions (illustrated as points in the figure) of the two vessels. The polylines formed by the positions of the vessels represent the geometries of their trajectories. The geometry of the red trajectory is associated with the tiles $c(i,j), c(i+1,j), c(i,j+1), c(i+1,j+1)$, while the geometry of the green trajectory is associated with the tiles $c(i,j), c(i+1,j), c(i,j+1)$.

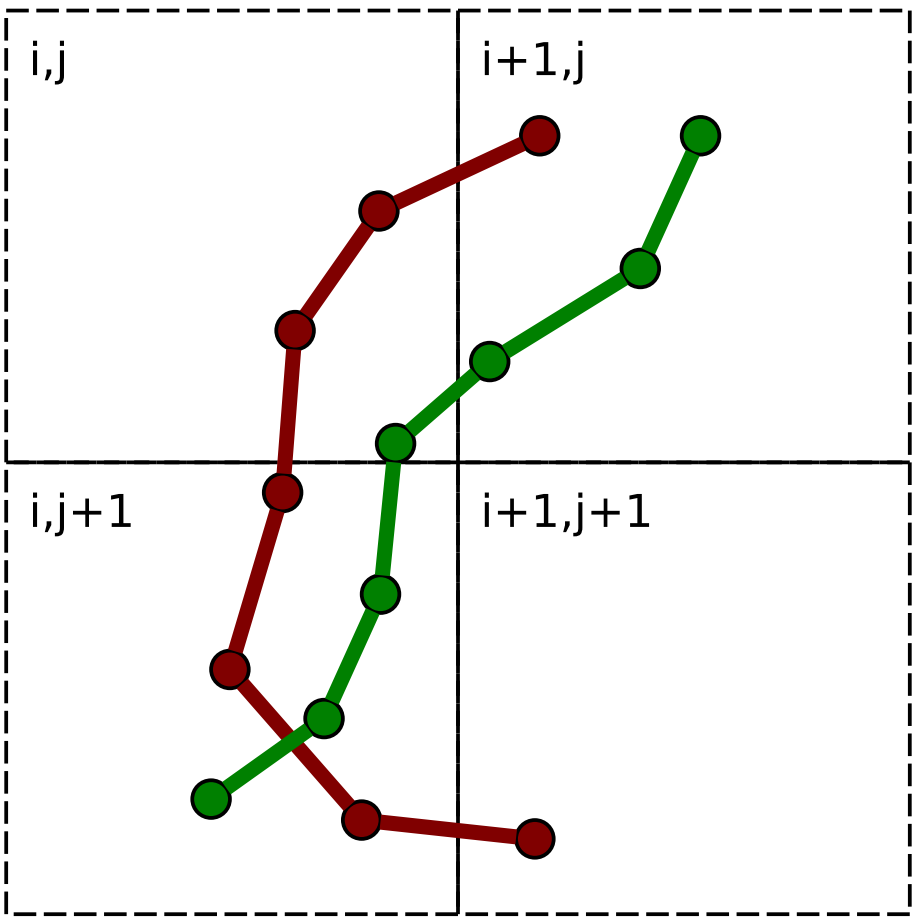


Fig. 3 Example of space-tiling of polylines using an equi-grid, where different colors indicate trajectories of different vessels

Special treatment is necessary in the case of proximity relations, where the aim is to find entities $s \in \mathcal{S}$ within distance θ from other entities $t \in \mathcal{T}$. In this case, we consider the *buffered geometry* g' of target entities, which essentially is a θ-extended geometry. Then, we can safely apply the procedure described above, to find the tiles $C(g')$ which contain g', without missing any candidate pairs.

In practice, in stLD, the space-tiling technique is first applied on each entity $t \in \mathcal{T}$ to populate the grid, and then applied on each entity $s \in \mathcal{S}$. However, the memory footprint of stLD can be reduced, when one-to-one relations are being computed. Specifically, one-to-one relations hold between exactly one entity of $\mathcal{S}$ and exactly one entity of $\mathcal{T}$, thus only one data set needs to be preserved in memory. The typical approach is to use the smallest data set as target $\mathcal{T}$ that is kept in memory, or the data set that is less likely to be modified in the future (e.g., coastline, protected areas, etc).

3.2.3 Refinement phase

By the end of the filtering phase, each geometry is associated with at least one tile. The refinement phase involves the processing of the entities in $\mathcal{T}$ using the same function $b(x, y)$ to identify the enclosing tiles for each geometry of the entities in $\mathcal{T}$. Tiles that are associated with at least two geometries provide the set of candidate pairs, i.e., for $t \in \mathcal{T}$ and a set $\{s_0, ...s_k\} \subseteq \mathcal{S}$ associated with the same tile, the set of candidate pairs are $\{(t, s_0), ..., (t, s_k)\}$. The processing entails a (possibly complex) geometrical test that has non-trivial computational cost, especially when applied on many candidate pairs.

The stLD engine supports a wide range of spatiotemporal relations, distinguished into proximity, topological and domain specific relations, between any type of geometry. Proximity relations, generally represented as $\langle s, t \rangle \in \mathrm{nearby}_\theta$, are typically computed on points or polylines (representing trajectories). In the latter case, the nearest points between the geometries s,t are calculated and checked if their distance satisfies the threshold $d(s, t) \leq \theta$. In the case of spatiotemporal proximity, where additionally the points must satisfy a temporal threshold, an additional constraint needs to be checked.

Specifically, a point-to-point proximity relation holds between the reported positions of two moving objects if and only if the spatial distance between their positions is less than a predefined distance threshold, and the difference between their timestamps is less than a given temporal threshold value. The distance between points is computed using the Haversine formula. This type of proximity relation is very common and widely used on various domains due to its simplicity.

One disadvantage of the point-to-point proximity relations is that relations can be only established between explicitly reported positions. However, the movement of vessels is continuous forming trajectories over a period of time, relations between entities may also exist between implicit positions of moving objects. In the example of Fig. 3, computing the distances between points in tile $i, j + 1$ could fail to detect a proximity relation for small distance threshold, even though the trajectories intersect.

The segment-to-segment proximity relation imposes the use of a polyline geometry for each moving object to represent its trajectory, formed by the reported positions of each entity (in temporal order). Similarly, the temporal constituent of entities is formed by a sequence of the time intervals defined by the corresponding timestamps of consecutive reported positions. For the evaluation of the segment-to-segment proximity relation, we

employ a function $d(u, v) = (p_u, p_v)$ which given two geometries *u*, *v* returns the pair of points (p_u, p_v), s.t. p_u (respectively p_v) is a point on the geometry *u* (respectively, *v*) and the Haversine distance between p_u, p_v is the minimum distance between the geometries *u*, *v*. The points p_u, p_v are annotated with the timestamps computed by linear interpolation. Let t_{u0} (respectively t_{v0}) the timestamp on the immediate preceding reported position p_{u0} (respectively p_{v0}) of p_u, (respectively p_v) and t_{u1} (respectively t_{v1}) the timestamp on the immediate successive reported position p_{u1} (respectively p_{v1}) of p_u (respectively p_v). The timestamps for p_u, p_v, t_u and t_v are computed by linear interpolation of values t_{u0}, t_{u1} (respectively, t_{v0}, t_{v1}), assuming constant speed (magnitude and direction) between positions p_{u0} and p_{u1} (respectively p_{v0} and p_{v1}). The segment-to-segment proximity relation holds, iff $H(p_u, p_v) \leq \theta_d$ and $|t_u - t_v| \leq \theta_t$, where $H(p_u, p_v)$ the Harvesine distance between the closest points p_u, p_v of geometries *u*, *v*, and θ_d, θ_t the spatial and temporal thresholds respectively.

3.2.4 Spatiotemporal link discovery for streaming data

The stLD approach on processing the data record-by-record, inherently supports streaming data. Each position of an entity reported in the stream of data, will be associated with a set of tiles and evaluated with positions of other entities already being associated in the same tiles for a given set of relations. In the case of proximity relations between moving objects however, it is necessary that the "active" positions are maintained in the grid for some period of time, i.e., those positions that have the potential to satisfy the constraints of the relation when evaluated with future data that will be received from the stream. In case of point-to-point proximity relations, it is sufficient to maintain exactly one position for each entity, while in segment-to-segment proximity relations a set of consecutive points for each entity is required. In any case, it is safe to remove from the grid all the positions that have a timestamp difference from the latest data read from the stream, greater than the temporal threshold set on the proximity relations (we name these positions as "obsolete" positions). A frequent elimination of obsolete positions results to processing overhead, since the entire grid in memory, should be scanned. Less frequent elimination of obsolete positions results to a waste of resources (memory and processing time), since more candidate pairs at refinement phase, that will not satisfy the constraints of the relations are accumulated.

The stLD approach on streaming data sources, employs a sliding window which enables the grid to maintain for a fixed time duration the data received from the stream. Formally, a sliding window is defined by its time duration w_d, and the time instance specifying its ending point w_e on the timeline (the starting point of the sliding window is computed by the difference $w_e - w_d$). The ending point of the sliding window is constantly being updated by the latest timestamp on the data received from the stream (which makes the window "slide" on the timeline). On streaming data processing, stLD computes the starting point of the sliding window, and considers as obsolete any positions that are past the starting point of the window. The duration of the sliding window may affect the results of relations, depending on their definition and implementation. For example, a duration of a sliding window smaller than the temporal threshold defined on proximity relations may affect the computed result set. Therefore, the duration of the sliding window has to be specified taking into consideration the relations being evaluated.

In addition to the above, the loosely coupled components CER and stLD independently process the stream of data and exchange messages when needed. The messages from CER to stLD are requesting information about relations of certain entities reported in the stream. From the temporal point of view, CER requests refer to a time interval within the current or a near-future update of the sliding time window. Apparently, some requests cannot be answered at the time they are received, due to the lack of data and the link discovery method has to tackle this issue. stLD extends the filtering process to allow the inclusion of requests along with entities in tiles of the grid. This extension makes requests persistent within the corresponding tiles and handled as "hypothetical" positions of entities. The relations between hypothetical and existing positions are evaluated during the refinement step, and any findings are included in the response for the request. Requests that are past the sliding window can be safely removed, since there can be no new data in the stream that can alter the results for those requests (i.e., the temporal constraints can no longer be satisfied). This extension to the filtering part enables the processing of multiple requests in a scalable way by stLD.

4 CER powered with stLD

The stLD component performs semantic spatio-temporal reasoning on raw stream data, whereas Wayeb performs reasoning at a higher abstraction level in order to detect interesting activity patterns. Our intention is to investigate how Wayeb could establish communication with the stLD engine so that it can take advantage of such an engine's reasoning capabilities. The result will be a high performance unified engine that can perform reasoning at various levels (see Fig. 4).

4.1 Complexity of event recognition

We first have a closer look at how Wayeb works internally. The workflow is the following (see also Algorithm 1). The user provides a pattern in the form of a SRE (symbolic regular expression) and the engine compiles this pattern into a SFA T. Subsequently it creates a streaming version of this SFA T_s. This streaming SFA is then fed with a stream S of simple events. The version of Wayeb that we will be using works with non-deterministic automata. Thus, for each pattern (and automaton), Wayeb maintains a set of active runs. This set is updated after every new event arrival and is denoted by $\mathrm{Run}(T_s, S_{..i})$, where $S_{..i}$ is the stream up to index i. Initially, before any input event has been consumed, the set of runs $\mathrm{Run}(T_s, S_{..0})$ is composed of a single run, $[1, T_s.q_s]$, i.e., the automaton's head points to the first index in the stream and the automaton is in its start state. The engine then reads input events one by one and updates its set of runs after every new event. At timepoint k, before reading t_k, it maintains the set $\mathrm{Run}(T_s, S_{..k-1})$. After the event t_k, it produces $\mathrm{Run}(T_s, S_{..k})$. This is achieved by evaluating t_k against every $\varrho \in \mathrm{Run}(T_s, S_{..k-1})$. Each run $\varrho = [1, q_1] \overset{\delta_1}{\rightarrow} [2, q_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k]$ (where δ_i are transitions) has to evaluate t_k on all the outgoing transitions of state q_k. If no transition is triggered, this means that the SFA cannot move to another state and it is thus discarded and removed from the set of runs. It will no longer be included in $\mathrm{Run}(T_s, S_{..k})$. If only one transition is triggered, then ϱ is updated, becoming $\varrho = [1, q_1] \overset{\delta_1}{\rightarrow} [2, q_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k] \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}]$, with a new

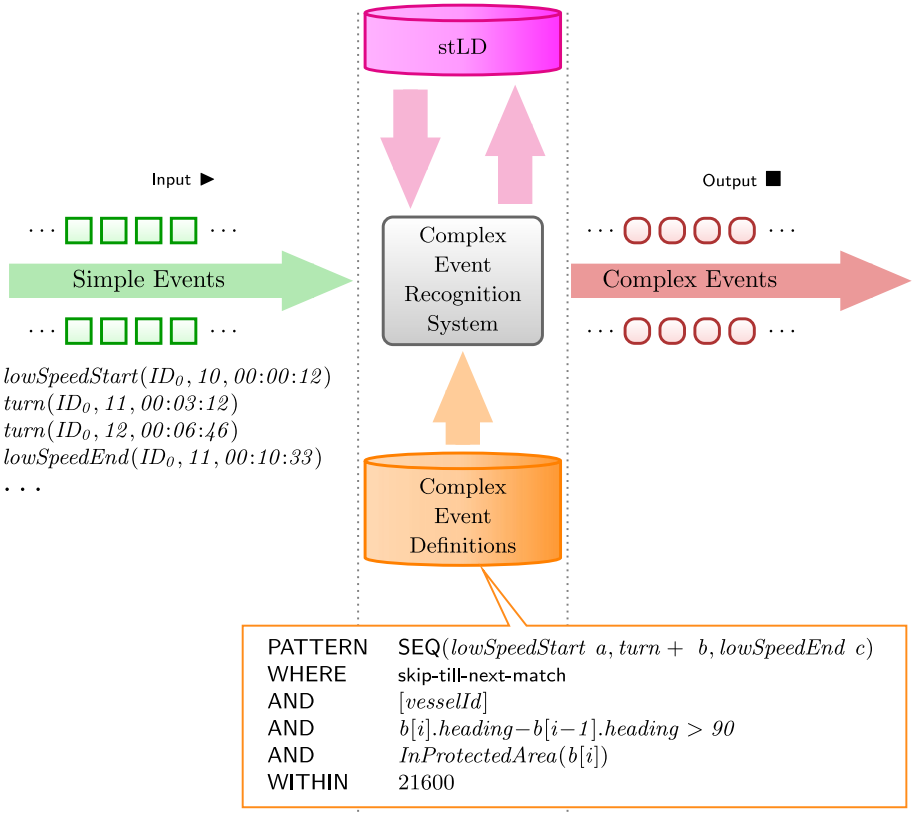


Fig. 4 Overview of CER and stLD. The CER system consumes a stream of annotated maritime events. Upon this stream, the system tries to detect instances of illegal fishing, defined as sequences of events where the vessel has low speed and executes one or more sudden turns. Additionally, we require that all turns are executed inside a protected area. The evaluation of the predicate of InProtectedArea is delegated to the stLD module which has a two-way communication with the CER system

state q_{k+1}. If n transitions are triggered and thus n next states are to be reached, then ϱ may be updated as usual for one of those next states. For each of the other $n-1$ next states, ϱ is first cloned, producing $n-1$ new runs ϱ', ϱ'', etc. Then each of these runs is updated with the new state and register contents

- $\varrho' = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k] \xrightarrow{\delta'_k} [k+1, q'_{k+1}]$
- $\varrho'' = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k] \xrightarrow{\delta''_k} [k+1, q''_{k+1}]$
- ...

The updated/new runs are added to the set of runs $\mathrm{Run}(T_s, S_{..k})$. Accepting runs are the exception here. If $q_{k+1} \in T_s.Q_f$ for some run ϱ, then ϱ reports that a complex event has been detected and is then "killed", i.e., not added to $\mathrm{Run}(T_s, S_{..k})$. This process is repeated for the remaining runs of $\mathrm{Run}(T_s, S_{..k-1})$.

Input: $SFA\ T_s$, input event t_k, active runs $Run(T_s, S_{..k-1})$
Output: Active runs $Run(T_s, S_{..k})$, accepting runs $Run_f(T_s, S_{..k})$

$Run_f(T_s, S_{..k}) \leftarrow \emptyset$;
$Run(T_s, S_{..k}) \leftarrow \emptyset$;
foreach $\varrho \in Run(T_s, S_{..k-1})$ **do**
  $C \leftarrow FindSuccessorStates(\varrho, t_k)$;
  if $|C| > 0$ **then**
    $c \leftarrow$ pick and remove element from C;
    $\varrho_{new} \leftarrow UpdateRun(\varrho, c)$;
    if $IsAccepting(\varrho_{new})$ **then**
      $ReportMatch(\varrho_{new})$;
      $Run_f(T_s, S_{..k}) \leftarrow Run_f(T_s, S_{..k}) \cup \varrho_{new}$;
    else
      $Run(T_s, S_{..k}) \leftarrow Run(T_s, S_{..k}) \cup \varrho_{new}$;
    foreach $c \in C$ **do**
      $\varrho' \leftarrow Clone(\varrho)$;
      $\varrho_{new} \leftarrow UpdateRun(\varrho', c)$;
      if $IsAccepting(\varrho_{new})$ **then**
        $ReportMatch(\varrho_{new})$;
        $Run_f(T_s, S_{..k}) \leftarrow Run_f(T_s, S_{..k}) \cup \varrho_{new}$;
      else
        $Run(T_s, S_{..k}) \leftarrow Run(T_s, S_{..k}) \cup \varrho_{new}$;
return $Run_f(T_s, S_{..k})$,$Run(T_s, S_{..k})$;

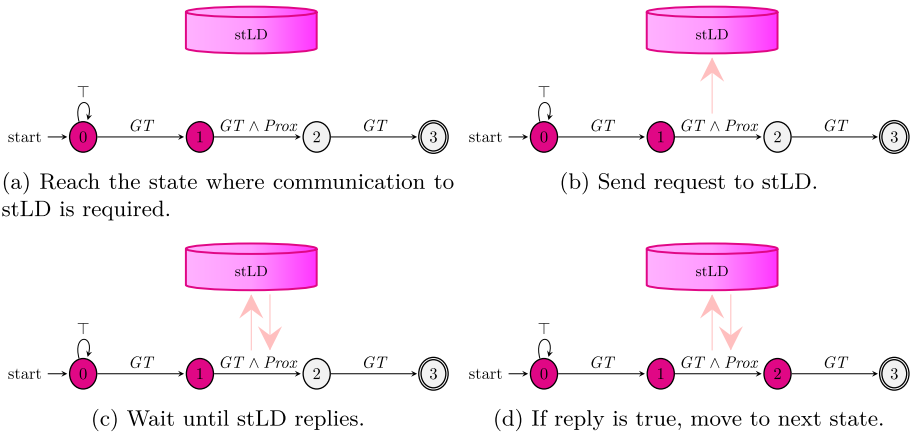
Algorithm 1 Running Wayeb with non-deterministic SFA.

There are two main factors that affect the performance of the CER engine. The first such factor is the evaluation of the Boolean expressions on the outgoing transitions of a run's current state. In cases where this evaluation has low complexity (e.g., checking whether the value of the speed attribute is above or below some threshold is an operation with constant, very low complexity), then moving a run to its next state(s) also becomes an operation which incurs low latency. The second factor is the number of active runs at each timepoint. Since each active run must be checked against every new incoming event, a proliferation of runs may have a significant impact on the latency of processing input events.

Our goal is to address the complexity issues arising from the first of the above factors. The reason is the following. In the maritime domain, it is often the case that an event pattern may be required to perform complex spatiotemporal tasks. For example, instead of checking simply for the speed of a vessel, it may be required to check whether a vessel is near any other vessel with a certain given time window. Such relations between vessels (or vessels and areas) are generally expensive to compute and would incur a performance penalty on the CER engine. On the other hand, they may be efficiently computed by specialized, optimized modules, such as the stLD. Thus, the opportunity arises for the CER engine to delegate the evaluation of these relations to the stLD. In order to do this, though, we need to ensure that the CER and stLD modules are able to communicate efficiently and that the communication cost is actually lower than the cost of evaluating such relations locally.

4.2 Communication between CER and stLD

The basic way for establishing a communication link between the CER and stLD modules is through a blocking scheme, as shown in Fig. 5. *GT* stands for *Greater Than* and is a simple



(a) Reach the state where communication to stLD is required.

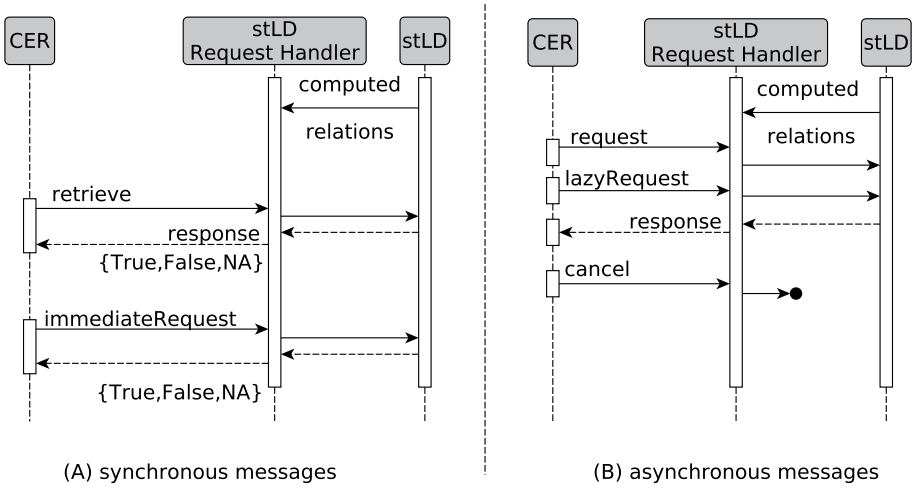
(b) Send request to stLD.

(c) Wait until stLD replies.

(d) If reply is true, move to next state.

Fig. 5 Blocking scheme

threshold predicate, determining whether the speed of the monitored vessel exceeds some given (not shown here) threshold. *Prox* stands for *Proximity* and determines whether the monitored vessel is close to any other vessel. *GT* is a simple predicate and can be directly evaluated by the CER engine, where *Prox* is substantially more complex (we need to locate all possible neighbouring vessels within certain spatial and temporal windows) and is handled by the stLD module. When an automaton run reaches a state, e.g., state 1 in Fig. 5, whose outgoing transitions contain stLD-related predicate, e.g., predicate *Prox* in Fig. 5, then the CER engine sends a relevant request to stLD and blocks, waiting for a reply — see the *immediateRequest* message in Fig. 6. As soon as the reply reaches back to the CER engine, the run may continue and determine its next state. In cases where a predicate is expensive, the CER engine sits idle waiting for the reply, despite the fact that could process other runs in the meantime (of the same or even of other patterns).



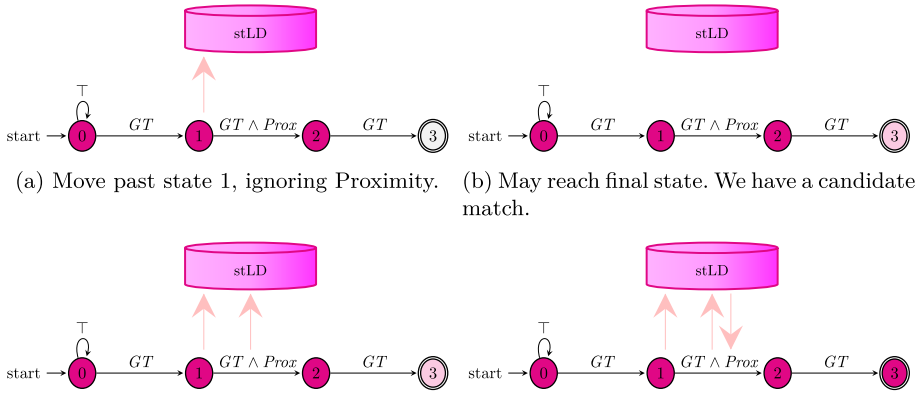
(A) synchronous messages

(B) asynchronous messages

Fig. 6 Messages exchanged between CER and stLD for blocking and lazy schemes

The stLD module implements internally a request handler for the communication with CER. Its primary role is to accept requests from CER. Furthermore, the request handler also serves as a cache for computed relations (i.e., a pair of vessels is related, if it satisfies a predicate) detected by stLD. This cache aims to minimize the delay between requests and responses. As stLD monitors the stream, it proactively transmits any computed relations to the request handler, even before any CER requests have arrived, as shown in Fig. 6. In case CER has a request about a relation that has been previously computed, the requestHandler instantly responds with the result. Otherwise, if a request involves a relation that has not been computed yet, the request handler issues a query to the stLD, and the latter will respond with either TRUE (if there are vessels satisfying the query) or FALSE (once stLD proves that no vessels can satisfy the query).

An alternative approach would be to employ an optimistic, lazy strategy. Whenever a run reaches a state which needs to communicate with the stLD, it can send a request but avoid waiting for the reply, as shown in Fig. 7 (see message *lazyRequest* in Fig. 6). Instead, we can make the optimistic assumption that the reply to the request is TRUE and move the run forward according to this assumption. For example, in Fig. 7 we may assume that Prox=TRUE and, if GT=TRUE as well, jump from state 1 to state 2. From state 1 we send an initial "request", but we do not block and wait for the reply. The stLD and CER modules can then work in parallel. While the stLD module evaluates a relation, the CER can still keep working on its runs. However, in this case, each run will also have a "debt" that it will need to pay at some point in the future. This debt corresponds to our optimistic assumption about the stLD-related predicate being true. For each such request sent to stLD, a run carries a corresponding debt, in the sense that, if the run actually reaches its final state at some point, then it will have to examine whether its optimistic assumption was actually sound. For example, if the run of Fig. 7 reaches state 3, we cannot immediately determine whether this is an actual match (complex event). We first need to check whether the Prox request sent from state 1 was actually TRUE. For this reason, when a run reaches a final state, it tries to repay its debt. For each stLD request it had sent, it now sends an equivalent "retrieve" message. If the stLD module has had enough time to evaluate the request, it will respond with the actual reply (which it holds in a special data structure). Otherwise, it sends a "NA"



(a) Move past state 1, ignoring Proximity.

(b) May reach final state. We have a candidate match.

(c) Now send a request to stLD.

(d) If reply is true, convert the candidate match into a real match.

Fig. 7 Lazy scheme

(not available) response. Finally, the CER engine checks whether its optimistic assumptions were actually valid, according to the actual responses of stLD. If they were, Wayeb validates and commits the run as an actual match. If any of the assumptions were wrong, the run is discarded. If some assumptions were correct and some replies were not yet availabe, the run repays the debt corresponding to the correct assumptions and retains the remaining debt. It then retries to repay this debt whenever a new event arrives.

The power of the lazy scheme lies in the fact that it allows the CER engine to keep working on runs while the stLD evaluates any received predicates. Thus, it offers a kind of parallel execution. The downside of this scheme is that it introduces a communication cost. Compared to the blocking scheme, each predicate evaluation requires at least an extra "retrieve" messages (besides the "request" and "reply" messages) and possibly multiple such messages in cases where the stLD module replies with "NA". Thus, it is possible that the lazy scheme might not always achieve a higher performance than the blocking one. The expectation is that it would be beneficial in cases where the cost of evaluating the remote predicate is significantly higher than the communication cost.

4.3 Predicted performance

In order to gain a better understanding of the effects of the lazy scheme on the performance of CER, we introduce and define some performance metrics. Let $P(t)$ denote the set of partial matches/runs maintained by the CER engine right after time index t. In other words, $P(t) = \mathrm{Run}(T_s, S_{..t})$, where T_s denotes the original streaming automaton and $S_{..t}$ is the stream until time index t. $P(t)$ is a more convenient way to represent the automaton runs. Let $C(t)$ denote the set of complete matches (i.e., complex events) right after time index t.

Latency given a set of partial matches at time t, $P(t)$, and a new event at $t+1$, S_{t+1}, latency is the difference between a) the time all new partial ($P(t+1)$) and complete ($C(t+1)$) matches are produced and b) the time S_{t+1} arrived at the system:

$$L(t+1) = T(P(t+1), C(t+1)) - T(S_{t+1})) \tag{1}$$

The above formula includes the evaluation cost.

Memory footprint memory footprint at time t (after event S_t has been processed), $M(t)$, is the memory occupied by the alive partial matches. We assume here that complete matches are immediately reported and discarded. However, we still need to compute them. We do not need to quantify the memory footprint of the complete matches during processing of S_t, since all complete matches $C(t)$ are derived from the previous partial matches $P(t-1)$. Thus, $M(t-1)$ reflects this.

A partial match p is a set, composed of input events. The size of the match $|p|$ is thus also a factor. We assume that a partial match p does not refer directly to input events ($p = \{S_{t_i}, \cdots, S_{t_j}\}$) but consists of indices to input events ($p = \{t_i, \cdots, t_j\}$). Thus, $|p| = |\{i, \cdots, j\}|$. The size of each individual event may also be a factor. In terms of bytes, this size depends on the number and type of its attributes. We assume that events come from the same schema and have the same size. Thus, $M(t) = \sum_{p \in P(t)} |p|$. A more realistic esti-

mation would have to take into account the fact that a partial match *p* needs to also contain the current automaton state. In fact, for the current version of Wayeb, a partial match/run is just an automaton state. Thus

$$M(t) = |P(t)| \tag{2}$$

Communication cost the communication cost at time *t*, *Com*(*t*), is defined as the total number of messages exchanged between the CER and stLD modules. We ignore the size of the messages since they contribute a constant factor. The communication cost is broken down into the outgoing cost, *ComOut*(*t*), and the incoming cost, *ComIn*(*t*). The former corresponds to the messages sent by the CER to the stLD module; the latter to those received from the CER by the stLD module. We can estimate these costs as follows. A partial match p is in a given state q, S_{t+1} arrives and we must evaluate all of q's outgoing transitions $q.\Delta$. Now, each $\delta \in q.\Delta$ might have multiple remote requests, since its guard might be a Boolean expression containing multiple remote predicates. Since all requests are of constant size, $ComOut(p) = \sum_{\delta \in q.\Delta} R(\delta)$, where $R(\delta)$ is the number of requests made by δ. *ComOut*(*p*) is the size (i.e., number) of all requests from match *p*, thus $ComOut(t+1) = \sum_{p \in P(t)} ComOut(p)$. Concerning the size of the replies, we expect binary answers from stLD (yes or no). Thus, $ComIn(t+1) = ComOut(t+1)$.

Based on the above analysis, we can make some rough predictions about the performance of the lazy scheme compared to the blocking one. For the purpose of bringing the lazy scheme into sharper contrast, we also consider another hypothetical communication scheme, called "eager". According to this scheme, we build a probabilistic model and determine, based on this model, the probability of a predicate evaluating to TRUE. If this is a high probability, we evaluate the predicate early, fetch the result and store it in a cache memory for fast retrieval. Notice that this scheme might not always be applicable. For example, in the maritime domain, we cannot pre-evaluate a *Proximity* predicate, since we do not know the vessel's coordinates at future timepoints.
For **latency**, we have the following, assuming that the latency of blocking is $L_{block}(t+1)$:

- Lazy: Typically, we should expect $L_{lazy}(t+1) < L_{block}(t+1)$. $L_{lazy}(t+1) > L_{block}(t+1)$ might happen in cases where many invalid candidate matches are generated and live longer than they should.
- Eager: $L_{eager}(t+1) < L_{block}(t+1)$. Improvement depends on when exactly it is decided to prefetch and the accuracy of the predictor (the underlying probabilistic model). Deciding early means that the predictions might be less accurate, which might offset any benefits due to earliness. If the predictor gives many false negatives, then we'll have a small reduction. In fact, it is conceivable that an inaccurate predictor who produces many false positives (keeps prefetching data elements that will not actually be needed) will incur an overhead. Here, we assume that the CER and prefetching processes are separate and run in parallel. Thus, the CER process would not be directly affected and we would not exceed $L_{block}(t+1)$. Still, if the cache is filled with irrelevant data, this could make $L_{eager}(t+1) - L_{block}(t+1)$ converge towards 0.

For **memory**, we have the following, assuming that the memory footprint of blocking is $M_{block}(t)$:

- Lazy: $M_{lazy}(t) \geq M_{block}(t)$. The reason is that extra matches might be created while waiting for the remote reply.
- Eager: $M_{eager}(t) = M_{block}(t)$. Since no extra partial matches are created, there is no memory overhead.

For the **communication** cost, we have the following, assuming that the communication cost of blocking is $Com_{block}(t+1)$:

- Lazy: $Com_{lazy}(t+1) > Com_{block}(t+1)$. The reason for the higher communication cost of the lazy scheme is twofold. First, besides the standard request/reply messages, the lazy scheme also requires extra messages, like "retrieve". In case of NA replies, multiple "retrieve" messages may be generated. Second, the lazy scheme typically produces more partial matches which live longer and would have been killed by the blocking scheme. These matches, allowed to run longer, might have extra "downstream" remote predicates requiring communication. It is interesting to note though that, with clever management (e.g., by using one reply to evaluate multiple candidate matches), the cost could be significantly lower.
- Eager: $Com_{eager}(t+1) \geq Com_{block}(t+1)$. The equality holds only if the predictor is perfect and data (or type) is prefetched only in cases where it will eventually be needed indeed. For false negatives, any (missing) requests will nevertheless be made eventually, so $Com_{eager}(t+1)$ can never be lower than $Com_{block}(t+1)$. But it can be higher, if the predictor produces false positives.

5 Empirical evaluation

We first present our experimental setup in Section 5.1. We then test the stLD and CER components independently. Section 5.2 presents results about the stLD component, whereas Section 5.3 is dedicated to the CER module. Finally, in Section 5.4, we test the system as a whole, where the CER module is aided by stLD for solving spatial tasks.

5.1 Experimental setup

The dataset used for the experimental evaluation spatially covers the area around the seaport of Brest, illustrated in Fig. 8 (different colors indicate different vessels) and temporally covers a period of six months (from October 2015 to the end of March 2016). The data set describes the movement of 5,055 distinct vessels in a total of 18,495,678 records. Each record in the data set reports an identifier of the vessel (MMSI), the navigational status, the rate of turn, the speed and course over ground, the true heading, the reported position (longitude, latitude in WGS84 reference system) and timestamp of the record [11, 32]. The stLD module was run using Oracle JDK 17.0.2 on a 8-core CPU at 2.1GHz with 8GB RAM. The CER component was run using Oracle JDK 1.8 on a 12-core CPU at 3.2GHz with 16GB RAM. For the communication between the two modules, Kafka 2.13 was used.

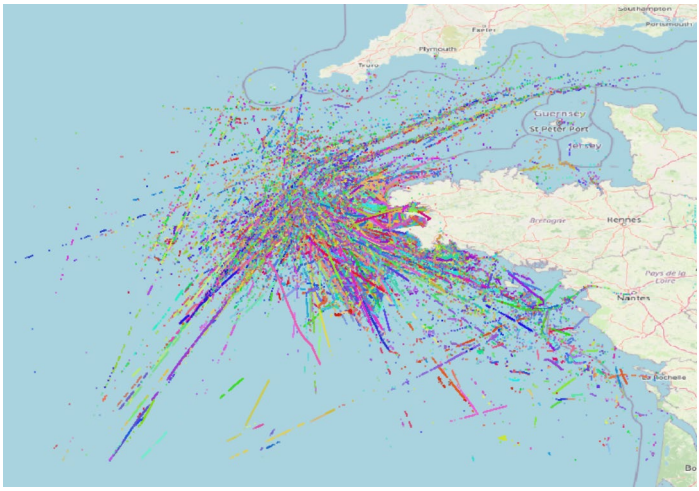


Fig. 8 Spatial coverage of the dataset used in the experimental evaluation

5.2 stLD experimental evaluation

In this section we evaluate the scalability of stLD on computing the point-to-point and segment-to-segment proximity relations, for distance thresholds $D\theta = \{10,100,500,1000\}$ meters and temporal thresholds $T\theta = \{60,600,1200\}$ (in seconds). The processing time for the point-to-point proximity relation is reported in Table 2. Specifically, the first column reports the distance threshold evaluated. The rest of the columns report the processing time (in milliseconds) and the computed relations for each temporal threshold applied. For example, the second and third columns report the processing time and the number of computed relations for the temporal threshold of 60 seconds. We observe that increasing the threshold values, results to more relations, which can be explained by the fact that relaxing the constraints, increases the possibility that a pair of candidates will satisfy the constraints, establishing a new relation.

In addition to the above, we report the processing time for various input sizes to evaluate the scalability of stLD on computing point-to-point proximity relations. Figure 9 illustrates the evaluated settings for input sizes varying from 50K to 450K (increment step 50K) records. The results show that the computation of point-to-point proximity relations scales linearly to the input size, while in these settings the threshold values do not seem to significantly affect the processing time.

Table 2 Point-to-point proximity: stLD processing time and number of relations for various distance and temporal thresholds

$D\theta$(m)	$T\theta$ 60 sec		$T\theta$ 600 sec		$T\theta$ 1200 sec	
	proc.Time (msec)	Relations	proc.Time (msec)	Relations	proc.Time (msec)	Relations
10	563551	122518	572185	180160	555033	185919
100	575072	1786913	573758	2597671	565097	2725205
500	557635	4651111	560462	8319956	552029	9210087
1000	558440	4751233	562592	8633787	553479	9552945

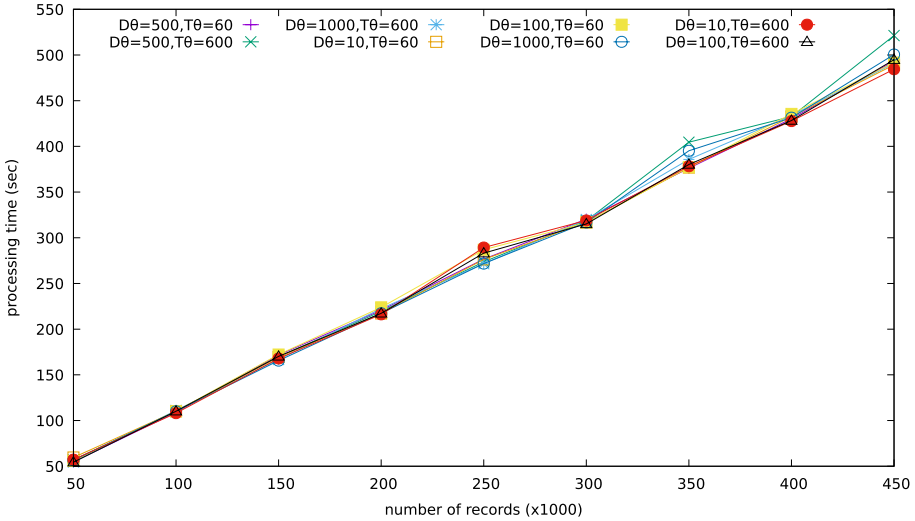


Fig. 9 The processing time on computing point-to-point proximity relations for various input sizes

Regarding the segment-to-segment proximity relations, it is expected that the size of the sliding window affects the overall performance of the link discovery process, since a larger window typically results in trajectory segments with larger length in the grid. Therefore, we report the processing time and the number of discovered relations for distance thresholds $D\theta = \{10, 100, 500, 1000\}$ in meters, temporal thresholds $T\theta = \{60, 600, 1200\}$ in seconds, and for the sizes of sliding window $w = \{600, 1200, 1800\}$ in seconds. Table 3 presents the experimental results. Specifically, the rows are grouped by window size, the first column reports the evaluated window size (in seconds), the second column reports the evaluated distance threshold, and each pair of columns that follow report the processing time and computed relations for each temporal threshold.

Table 3 Segment-to-segment proximity: stLD processing time and number of relations for various distance and temporal thresholds

Window size (sec)	$D\theta$ (m)	$T\theta$ 60 sec		$T\theta$ 600 sec		$T\theta$ 1200 sec	
		proc.Time (msec)	Relations	proc.Time (msec)	Relations	proc.Time (msec)	Relations
600	10	30743345	5212	32796507	8636	33482527	9083
	100	15760752	20786	15441838	25754	15447663	26604
	500	13660675	27328	12980068	33070	12937658	34346
	1000	13635508	27541	12967252	33423	13039539	34726
1200	10	86079268	4651	81793651	8505	75925272	10157
	100	35988996	18769	33045856	25338	31917509	27032
	500	29020256	25512	26215564	32431	24749118	34758
	1000	28894374	25715	26040563	32749	24841769	35133
1800	10	168338266	4301	151019866	8307	141277934	10168
	100	65303108	17264	57779422	24701	54800431	26827
	500	50444717	24022	42875068	31657	39859138	34434
	1000	50197616	24214	42480387	31968	39346402	34782

The experimental results confirm that the larger the window size for the segment-to-segment proximity relation, the higher the processing time. This can be explained by the fact that larger window sizes maintain and update longer trajectory segments (constructed by several reported positions). In general, such segments are expected to affect both the filtering (i.e., the longer the segments, the more tiles are excepted to be associated with) and the refinement processing. Interestingly, we observe that the longer the window size, the smaller the number of computed relations. By construction of the segment-to-segment proximity relation, the process returns one relation for each pair of segments satisfying the relation constraints. Thus, longer windows force a single relation between segments which would be partitioned into smaller segments (resulting to more relations) when shorter windows are used. The experimental results in these settings indicate that increasing the spatial threshold results to a considerable decrease of the processing time. We also observe that distance threshold seems to have the major impact on processing time in all cases. These observations can be explained by the construction of the segment-to-segment proximity relation. Please recall that in segment-to-segment proximity relation link discovery, an update from the stream of data, extends the trajectory of a moving object with a segment defined by the last reported position of the object and the newly reported one. The filtering process will indicate the tiles associated with the trajectory and the pairs of candidates will be generated. By construction, the nearest points between trajectory segments (of different moving objects) may not change with such an update on the grid. Thus, the same pair of candidates may be found in more than one updates on the grid. When some pairs of candidates that have been previously evaluated and lead to a relation, appear in the refinement step, they are excluded from processing, which affects the overall processing time.

Finally, we evaluate the processing time on computing the segment-to-segment proximity relations for the input sizes from 50K to 450K records (increment step is 50K). Figure 10 illustrates the results for window sizes $w = \{1200, 600\}$ (in seconds) and threshold setting $D\theta = \{100, 500\}$ in meters, and temporal thresholds $T\theta = \{60, 600\}$ in seconds. It is confirmed that a long window results in poor scalability, especially when combined with small distance threshold. This can be explained by the fact that the long trajectory segments (that are computationally expensive) fail to provide a relation at the refinement step, due to the small distance threshold, and they are re-evaluated in future updates of the sliding window.

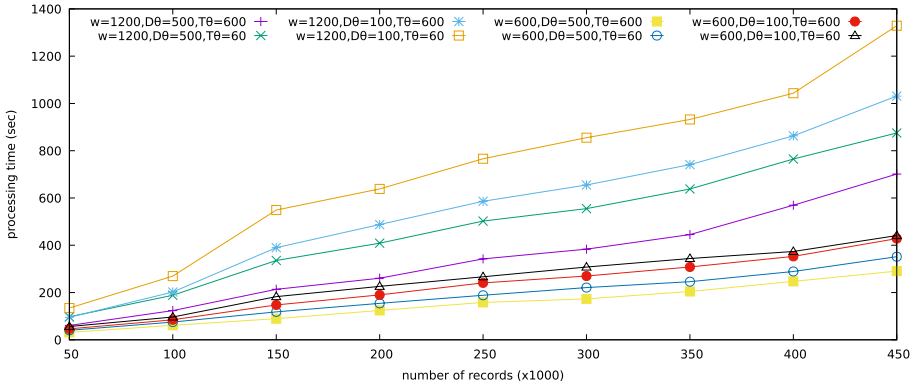


Fig. 10 The processing time on computing the segment-to-segment proximity relations on various input sizes

An "intermediate" class of cases can be distinguished, where a long window but also a large distance threshold is used. In this set of cases, the large distance threshold allows more pairs of segments to be related, thus they are never again considered as candidates in the refinement step. The large distance threshold counter-affects the cost of maintaining larger segments as a result of wider sliding windows. Finally, the best scalability results are provided by the settings where the window size is equal to the temporal threshold ($w = T\theta$). For example, we observe that curves reporting the settings where w=$T\theta$=600, are below the curves for w=600, $T\theta$=60 and the corresponding distance thresholds, for all input sizes. In these settings, the narrow window size results to smaller trajectory segments maintained in the grid, while the updating of the sliding window eliminates all the positions that will not satisfy the temporal threshold (i.e., if two segments satisfy the distance threshold, they will always be related).

5.3 CER experimental evaluation

An important feature of Wayeb is that it can detect patterns with low latency. It can thus scale gracefully with increased loads. We tested Wayeb's scalability in the maritime domain as a standalone component. Specifically, we used a real–world dataset composed of a set of trajectories from ships sailing at sea, emitting AIS (Automatic Identification System) messages that relay information about their position, heading, speed, etc, described above, in Section 5.1. The dataset is publicly available [32].

As a test pattern, we used one that detects speed violations, i.e., consecutive messages where a vessel's speed exceeds some given threshold, as in Fig. 1. This pattern is applied on a per-vessel basis, i.e., each vessel has its own "copy" of the pattern. We started by running this single pattern on the dataset. In order to increase the load of the engine, we then started increasing the number of patterns (by copying the original pattern multiple times). Note that each pattern (or copy thereof) is applied on all vessels. The number of running automata is thus equal to the number of pattern copies multiplied by the number of vessels. The throughput starts from approximately 1.000.000 events/second for a single pattern and decreases as the number of patterns increases. In Fig. 11 we show results for the scalability test. Instead of showing directly throughput numbers, we show the degree to which our system is better than the real-time requirements of the problem. The x axis corresponds to the number of patterns. The y axis corresponds to the ratio of execution over real time. The execution time is the total time that the engine required to process the whole dataset. By real time, we refer to the total time that elapses in the real world for the dataset to be produced, i.e., the difference between the timestamp of the last event in the dataset and the timestamp of the first event. A value of 1.0 for this ratio would mean that the engine can process events at the rate that these are produced. Any value above 1.0 would mean that the engine lags behind the input event and cannot process them in time. Any value below 1.0 would mean that the engine can cope with the event rate. Note that Wayeb, in all cases, can process the dataset at a rate that is orders of magnitude greater than the input event rate. As expected, the ratio increases as the number of patterns increases as well. However, it is always below 1 %, even with 100 patterns. This indicates that Wayeb can handle real-world maritime patterns with exceptional efficiency, even for increased workloads. More extensive results on the performance of Wayeb as a standalone module may be found in [26–28]. In [26], it was shown that the window size constitutes the most important factor affecting Wayeb's performance. For

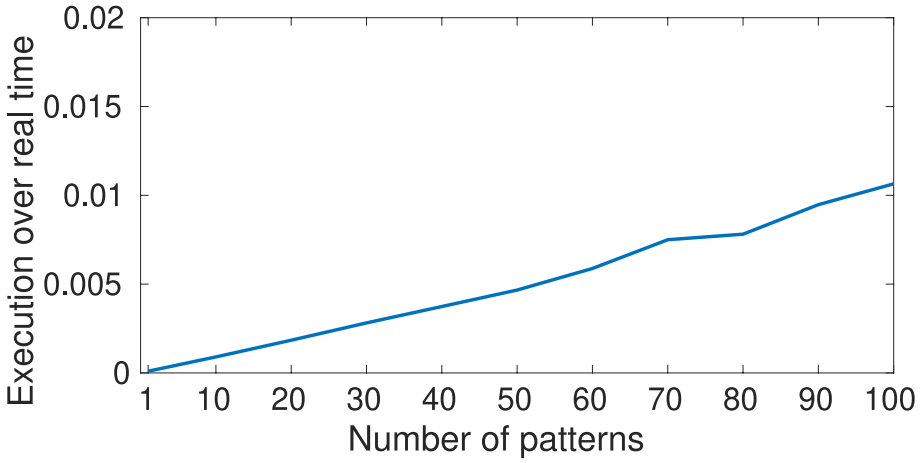


Fig. 11 Ratio of execution over real time as a function of the number of patterns

large windows, though, the pattern length also starts having an impact. In the general case, it is typical for Wayeb to exhibit a throughput performance reaching millions of processed events per second for various datasets and patterns.

Subsequently, we tested Wayeb against another type of workload variation: the input event rate. We applied an interpolation scheme on the initial dataset to produce derivative datasets where the interval between any two consecutive position signals of a vessel is fixed. We created datasets where the interpolation interval is 1, 5, 10, 15, 30 seconds, corresponding to a frequency of 60, 12, 6, 4, 2 events per minute for each vessel. The relevant results are shown in Fig. 12. We observe that, again, Wayeb can process all workloads at a speed which is orders of magnitude greater than the event rate.

5.4 CER - stLD experimental evaluation

We then compared the performance of the lazy versus the blocking scheme for various values of the communication and evaluation cost/delay. We used a pattern detecting a sequence of AIS messages where a vessel has a high speed, and it is close to another vessel:

$$
\begin{aligned}
&R_{prox} := x \cdot y \cdot z \text{ WHERE} \\
&\quad GT(x, speed, 5.0) \ , \\
&\quad (GT(y, speed, 5.0) \wedge ProximityRemote(y)) \ , \\
&\quad GT(z, speed, 5.0) \\
&\quad \text{PARTITION BY } vesselId
\end{aligned}
$$

The pattern is a sequence of three events. The constraint for the first and last events is the same: the speed must be greater than 5 knots. For the middle event, there is the extra constraint that the vessel must be in close proximity to at least another vessel. The suffix *Remote* indicates that the proximity predicate is evaluated by the stLD component.

For a systematic evaluation, we simulated the evaluation (of stLD) and communication delays. We also used a parameter to simulate the probability of stLD evaluating a predicate

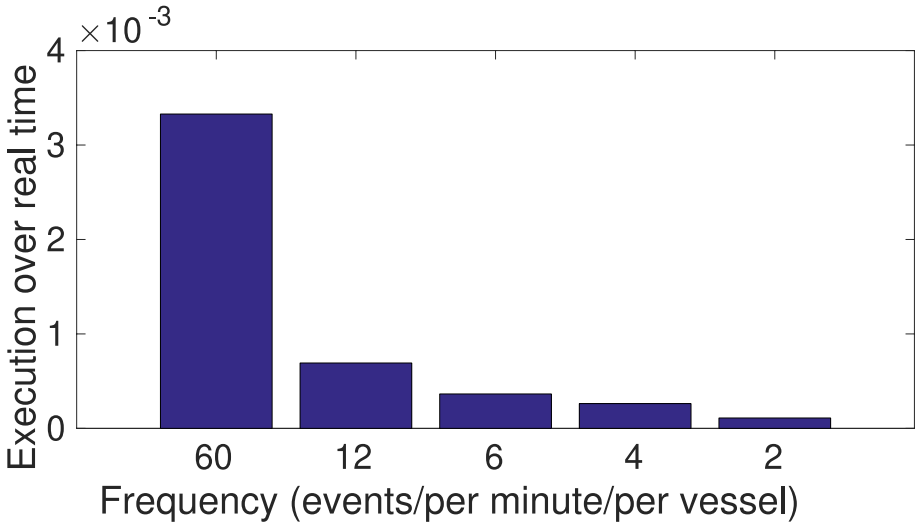


Fig. 12 Ratio of execution over real time as a function of input event frequency

as TRUE. Figure 13 shows throughput as a function of communication and evaluation delay. It also shows the maximum possible throughput with blocking, assuming that the latency of Wayeb is 0 (displayed as max). Figure 14 shows the throughput increase of lazy communication over blocking. We can see that lazy is almost always better, except for very low (evaluation and communication) delay values. Figure 15 shows again throughput as a function of communication and evaluation delay. However, this time we also run experiments for different values of satisfaction probability of the proximity predicate. The lower the

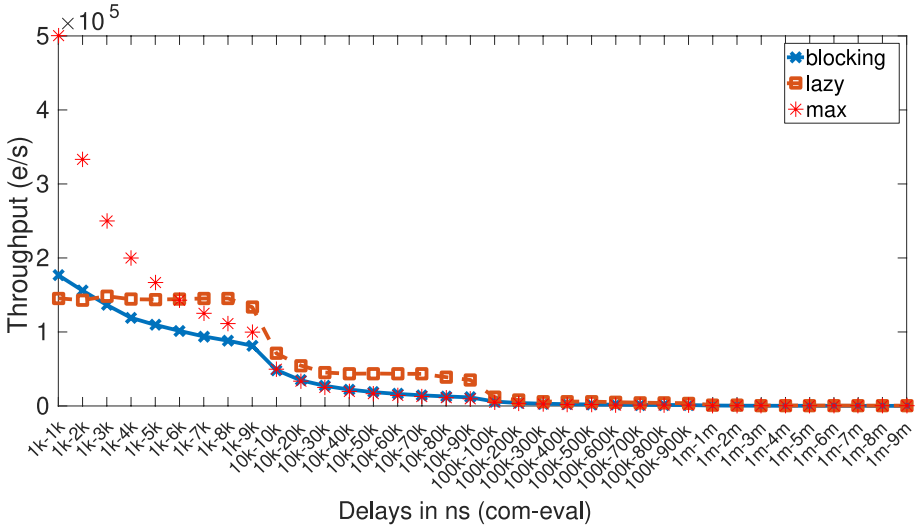


Fig. 13 Throughput as a function of communication and evaluation delay

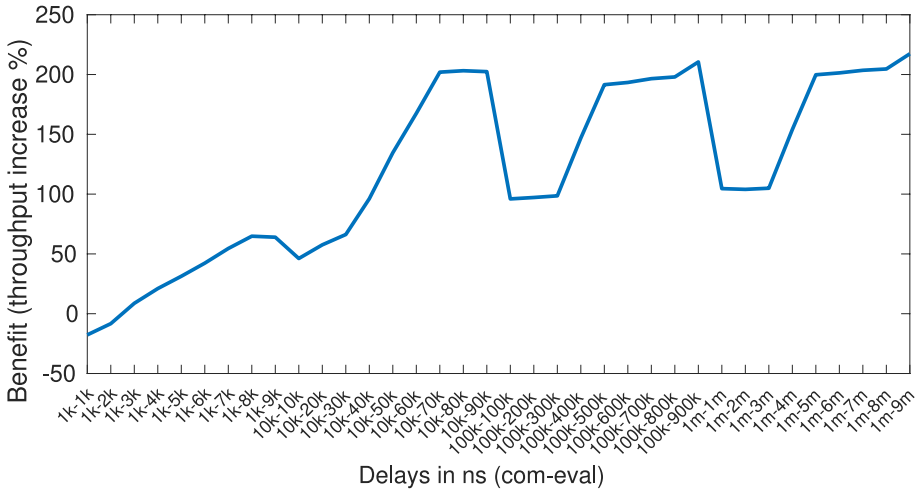


Fig. 14 Throughput increase of lazy over blocking as a function of communication and evaluation delay. Negative values indicate that the blocking scheme is actually better

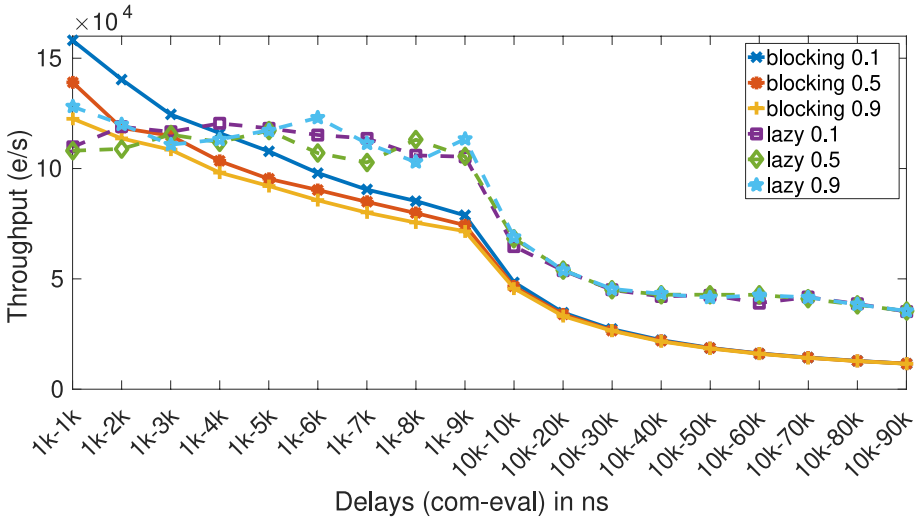


Fig. 15 Throughput as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temporal predicate. We plot the performance of Wayeb for both blocking and lazy, for three different values of the satisfaction probability (0.1, 0.5, 0.9)

value of this satisfaction probability the fewer the time that the stLD will reply with TRUE to a request from the CEP engine.

In Fig. 13, the drop in throughput from the max curve to the blocking curve for a given combination of delay values corresponds to the part of throughput that we lose due to the delay of Wayeb. Note that throughput is estimated as:

$$\frac{number\ of\ events}{WayebDelay + comDelay + evalDelay}$$

The total delay in the denominator is equal to the sum of the latencies incurred by all the input events, i.e., for a stream $S_{..t}$, we have

$$totalDelay = WayebDelay + comDelay + evalDelay = \sum_{1}^{t} L(t)$$

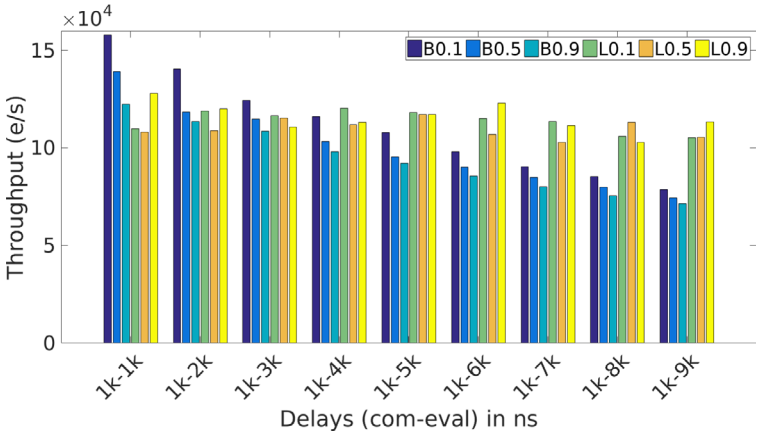
For example, for the x point at 1k-2k, the max throughput is 350.000 events/sec and the blocking throughput is 150.000 events/sec. Wayeb thus costs us 200.000 events/sec. Notice that the max and blocking curves start to coincide after a certain point. This means that the total delay (the sum of communication, stLD and Wayeb delays) is dominated by com-eval delays after that point and the delay of Wayeb is minimal compared to the com-eval delays. The denominators in the definition of throughput for both cases tend to become equal, which can happen only if $\mathrm{WayebDelay}$ is very small compared to the other two components (remember that $\mathrm{WayebDelay} = 0$ for the max case). Finally, it is interesting to note that lazy does not seem to be significantly affected by the evaluation delay.

We can also see in Fig. 14 that throughput increase of lazy over blocking may reach values of 200%. As initially suspected, the lazy scheme is most beneficial when the evaluation delay is large relative to the communication delay. Since the lazy scheme suffers from a higher communication cost, we need the evaluation cost to be relatively large. Otherwise, any benefits gained from our optimistic lazy scheme would be offset by the communication cost.

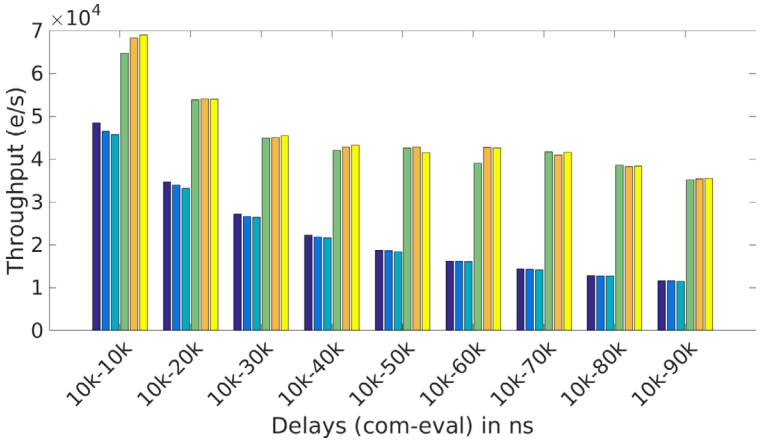
In Fig. 15 we see that the lazy scheme is not as much affected by the satisfaction probability as the blocking scheme. We also see that, as we decrease the satisfaction probability, the blocking scheme turns out to be better than the lazy for a wider range of delay values. The reason is that a lower satisfaction probability means that more runs are killed because of more FALSE replies from stLD. As a result, the lazy scheme advances more runs that it should not have. Therefore, we pay a higher overhead cost.

Figure 16 presents results from a different perspective, as bar charts. For each subfigure, we maintain a constant communication delay. Figure 16c presents additional results, with even higher values for the delays, where the communication delay is kept at 100*kns*. For each bar chart, we keep the communication delays constant on the *x* axis and vary the evaluation delay. We see again that the lazy scheme outperforms the blocking in most cases. As expected, the more we increase the evaluation delay (relative to the communication delay), the more pronounced the benefit for the lazy scheme becomes. It is also interesting to notice the behaviour of the two schemes for high delay values (see, e.g., Fig. 16c). For high delay values, the satisfaction probability has a negligible effect on throughput. The explanation behind such a behaviour is that the extra overhead for handling and killing runs (valid in the case of blocking and both valid and invalid in the case of lazy) becomes insignificant compared to the very high cost of the delays.

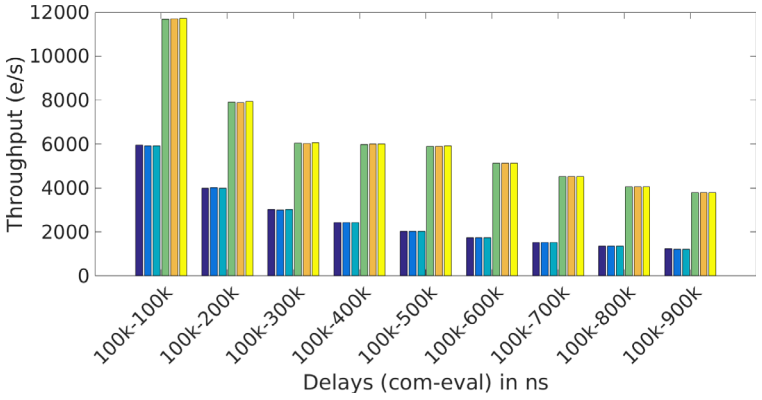
Figures 17 and 18 present two more variations of the behaviour of our system for various delay values. The setting is the same as that for the results shown in Fig. 16. The difference is that we increase the number of remote predicates in the tested pattern. We inject



(a) Communication delay is kept constant at $1k$ ns.



(b) Communication delay is kept constant at $10k$ ns.



(c) Communication delay is kept constant at $100k$ ns.

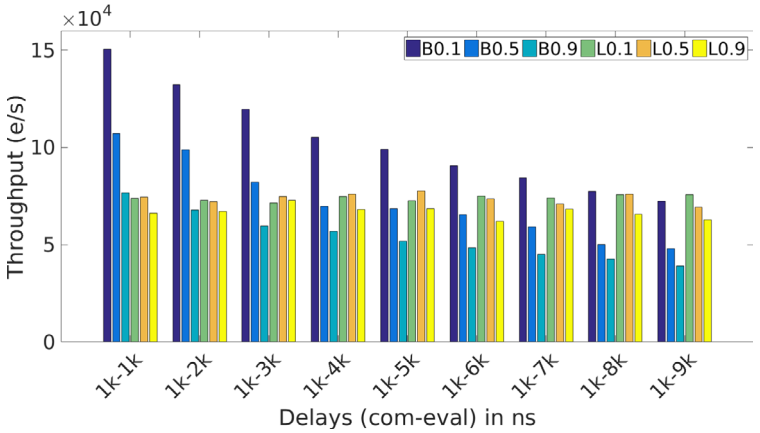
Fig. 16 Throughput as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temporal predicate. B: blocking, L: lazy. The pattern has one remote predicate

two remote predicates for the results of Fig. 17 and three for Fig. 18. In both cases, the general behaviour remains the same, i.e., the lazy communication scheme outperforms the blocking one. There are some intriguing differences, though. For example, for patterns with more remote predicates, there are more delay combinations for which the blocking scheme remains better, especially for low delay values (see Figs. 17a and 18a). This is due to the fact that patterns with more remote predicates are longer and tend to create more runs. Thus, the overhead of run handling remains significant for a wider range of delay values. Finally, it is interesting to notice that the lazy scheme remains more robust for various values of the communication delay. For example, in Fig. 18b we can observe how the performance of the blocking scheme for $\mathrm{threshold} = 0.1$ (dark blue bar) drops as we increase the communication delay. This is expected, as the engine needs to wait for more time whenever it blocks and waits for a reply from stLD. On the other hand, the performance of the lazy scheme for the same threshold value remains almost stable. For patterns with one remote predicate, this is not always the case (see, e.g., Fig. 16c). The reason is that, for longer patterns, the number of "NA" (not available) replies from stLD drops. This is how the evaluation delay may affect the performance of the engine. The runs are processed in parallel with the evaluation of stLD requests. The evaluation delay may thus have an impact on the throughput when the stLD sends "NA" replies instead of final responses. When a pattern has more remote predicates and is thus longer, each run takes longer to reach its final state. As a result, stLD has enough time to process the request and immediately send a final response when the run asks for one.

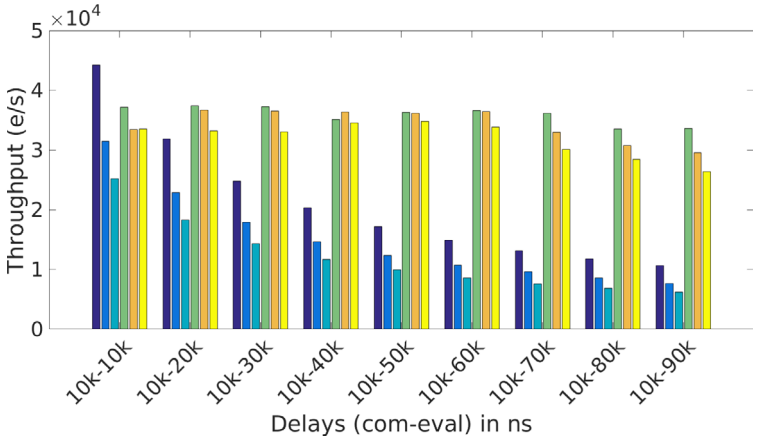
The above results confirm our initial assessment regarding the predicted latency of our system when using the lazy scheme (see Section 4.3). We predicted that $L_{lazy} < L_{blocking}$ for most cases and this is indeed the case. We also noted that the lazy scheme may become less efficient where many invalid candidate matches are generated. As shown in the results, this is also the case. For lower values of the satisfaction threshold, the system allows more invalid partial matches to remain alive, i.e., matches that would not have been generated by the blocking scheme. It is precisely for those low threshold values that we experimentally see the blocking scheme outperforming the lazy one.

We finally conducted experiments to investigate the memory footprint and the communication cost of each scheme. Figure 19 shows the total number of runs created in each experiment with a pattern of three remote predicates, which is indicative of the consumed memory (see Section 4.3). As expected, the number of runs increases as the satisfaction threshold increases. However, the difference between the two schemes is very small, with the lazy scheme exceeding the blocking one very slightly, thus only marginally confirming our initial prediction. Therefore, the number of runs does not seem to be an important factor explaining the difference in performance.

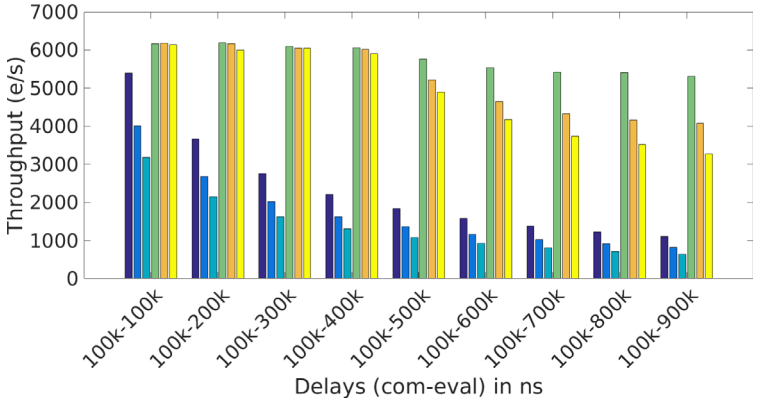
Figure 20 shows the total number of messages exchanged between the CER and stLD modules. For the blocking scheme, this number is equal to the sum of "requests" and "replies". For the lazy scheme, it is equal to the sum of "requests", "replies" and retrievals (see Section 4.2). As predicted, the communication cost is significantly higher for the lazy scheme. The two results about the number of runs and the number of messages indicate that, in cases where the blocking scheme is more efficient, this is not mainly due to the fact that runs are more numerous, but because they live longer. For the blocking scheme, we can observe that the communication cost becomes higher as the satisfaction threshold increases. This behaviour is due to the fact that an increased threshold allows more runs to survive. A



(a) Communication delay is kept constant at $1k$ ns.

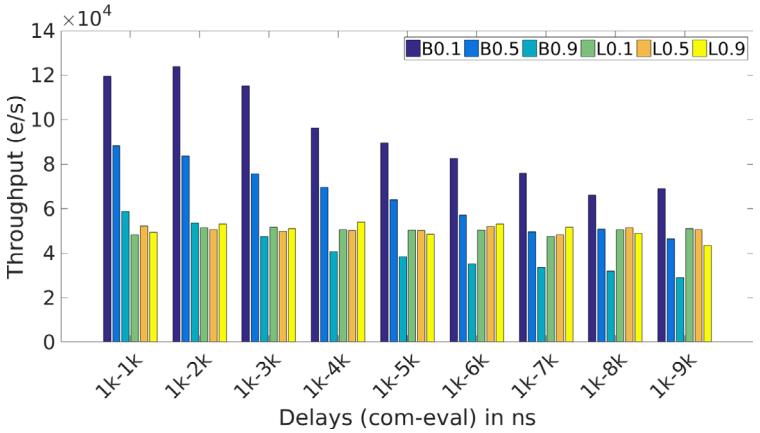


(b) Communication delay is kept constant at $10k$ ns.

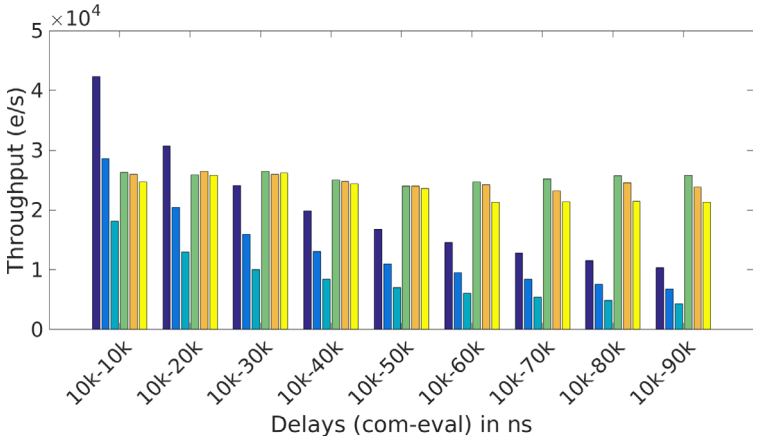


(c) Communication delay is kept constant at $100k$ ns.

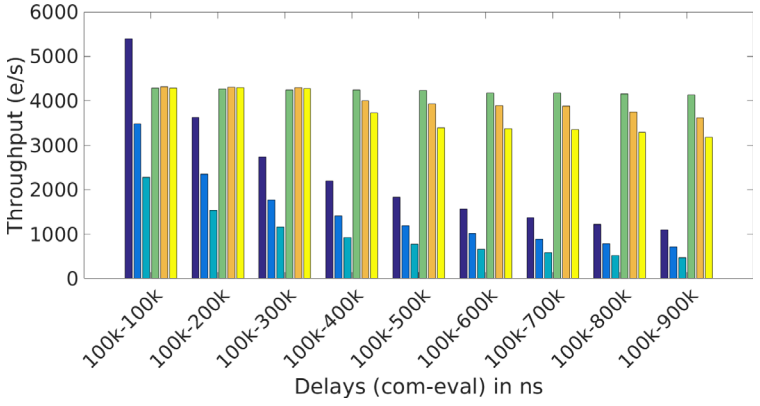
Fig. 17 Throughput as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temproal predicate. B: blocking, L: lazy. The pattern has two remote predicates



(a) Communication delay is kept constant at $1k$ ns.



(b) Communication delay is kept constant at $10k$ ns.



(c) Communication delay is kept constant at $100k$ ns.

Fig. 18 Throughput as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temproal predicate. B: blocking, L: lazy. The pattern has three remote predicates

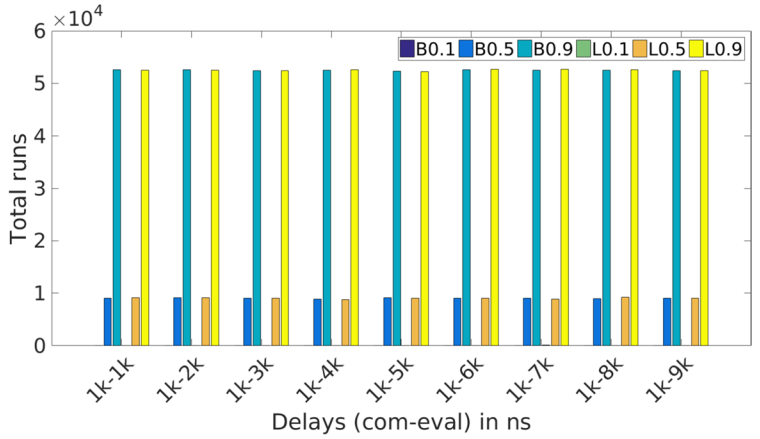


Fig. 19 Number of runs as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temporal predicate. B: blocking, L: lazy. The pattern has three remote predicates

run which survives the first remote predicate can then move on, reach the second one and communicate with the stLD module again. Interestingly, this is not the case for the lazy scheme. The number of messages is the same for all threshold values. The reason is that the lazy scheme is optimistic and never kills runs immediately. It rather allows all of them to reach their final state, regardless of the satisfaction threshold. The final "cleaning up" is only performed when the run reaches its final state. This indicates that there is room for optimizing the communication between stLD and CER by allowing runs to ask the stLD module for the final reply earlier. This is a line of work we intend to investigate in the future (Fig. 20).

6 Summary and further work

We presented an approach for integrating spatio-temporal background knowledge within a CER system. We demonstrated how a CER engine can take advantage of a dedicated spatio-temporal module in order to leverage its capabilities and maintain high throughput values even in the presence of computationally demanding spatio-temporal constraints. We described two different communication schemes between the CER engine and the spatio-temporal module: blocking and lazy. Our results showed that the lazy scheme is preferable in most cases. However, there are a few cases where the cost of maintaining an increased number of pattern runs is greater than any benefits gained from the lazy scheme and the blocking one turns out to be more advantageous.

As future work, at system level, we intend to explore more communication schemes. For example, currently, we wait until the automaton has reached a final state and then start sending retrieve messages to stLD. But we could do this in earlier states and thus limit the number of irrelevant runs. Alternatively, stLD could assume a more proactive role and send notifications to CER. Then CER could consume replies "immediately" and try to pay debts Communication schemes which attempt to prefetch any relevant information could be of interest as well. We also intend to automate the process of selecting the proper communi-

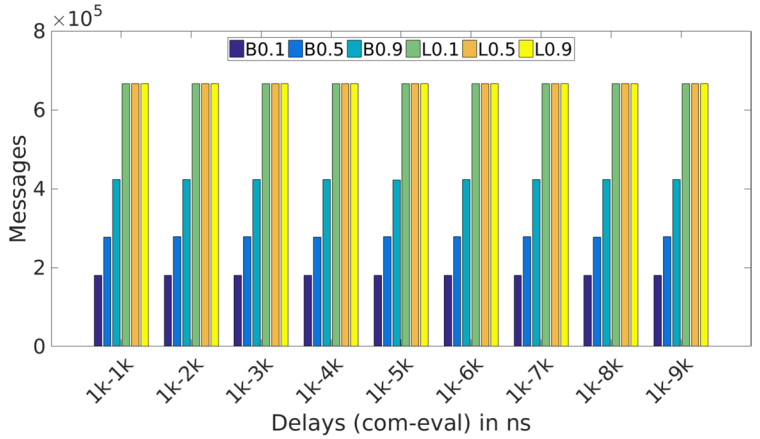


Fig. 20 Number of messages exchanged between CER and stLD as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temproal predicate. B: blocking, L: lazy. The pattern has three remote predicates

cation scheme. In each state, an automated system should be able to decide whether to, a) prefetch any relevant information; b) block and evaluate predicates; c) postpone, assuming the predicates are TRUE; d) or even postpone, assuming the predicate is FALSE, which implies that in cases where the predicate is evaluated finally as TRUE, we could possibly miss some complex events.

In terms of spatio-temporal reasoning, future work could focus on optimizing the processing of spatio-temporal relations in a streaming context. This is a challenging problem, especially for complex operations that include spatio-temporal joins, which is intensified by the streaming nature of input data and the strict latency requirements.

Moreover, another interesting extension would be to explore the integration of geospatial operations based on standardization efforts, such as the ones proposed by the Open Geospatial Consortium (OGC). In particular, the OGC API Moving Features is quite relevant to our work, as it is used for querying and accessing geospatial data that changes over time, such as vessel trajectories.

Author Contributions All authors (Elias Alevizos, Georgios Santipantakis, Christos Doulkeridis, Alexander Artikis) contributed to the design of the experiments, the writing of the paper and its revisions. Elias Alevizos and Georgios Santipantakis performed the experiments.

Funding This work was supported by the VesselAI project, which has received funding from the European Union's Horizon 2020 research and innovation program, under grant agreement No 957237.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Ethical Approval Not applicable.

Competing interests The authors declare no competing interests.

References

1. Giatrakos N, Alevizos E, Artikis A, Deligiannakis A, Garofalakis MN (2020) Complex event recognition in the big data era: a survey. VLDB J 29(1):313–352
2. Luckham DC (2005) The Power of Events - an Introduction to Complex Event Processing in Distributed Enterprise Systems. ACM, NY
3. Cugola G, Margara A (2012) Processing flows of information: From data stream to complex event processing. ACM Comput Surv 44(3):15–11562
4. Etzion O, Niblett P (2010) Event Processing in Action. Manning Publications Company, NY
5. Hedtstück U (2017) Complex Event Processing: Verarbeitung Von Ereignismustern in Datenströmen. Springer, Berlin
6. Artikis A, Zissis D (eds) (2021) Guide to Maritime Informatics. Springer, Switzerland
7. Idiri B, Napoli A (2012) The automatic identification system of maritime accident risk using rule-based reasoning. In: SoSE, pp 125–130. IEEE, Genova, Italy
8. Terroso-Saenz F, Valdés-Vela M, Skarmeta-Gómez AF (2016) A complex event processing approach to detect abnormal behaviours in the marine environment. Inf Syst Frontiers 18(4):765–780
9. Snidaro L, Visentini I, Bryan K (2015) Fusing uncertain knowledge and evidence for maritime situational awareness via markov logic networks. Inf Fusion 21:159–172
10. Patroumpas K, Alevizos E, Artikis A, Vodas M, Pelekis N, Theodoridis Y (2017) Online event recognition from moving vessel trajectories. GeoInformatica 21(2):389–427
11. Bereta K, Chatzikokolakis K, Zissis D (2021) Maritime reporting systems. In: Artikis A, Zissis D (eds) Guide to Maritime Informatics, pp 3–30. Springer
12. Santipantakis GM, Glenis A, Doulkeridis C, Vlachou A, Vouros GA (2019) stLD: Towards a spatio-temporal link discovery framework. In: Proceedings of the international workshop on semantic big data, SBD@SIGMOD 2019, pp 4–146. ACM, Amsterdam, The Netherlands
13. Alevizos E, Santipantakis GM, Doulkeridis C, Artikis A (2024) Online integration of spatial reasoning in complex event recognition. In: Proceedings of the workshops of the EDBT/ICDT 2024 joint conference, March 25, 2024. CEUR Workshop Proceedings, vol 3651. CEUR-WS.org, Paestum, Italy
14. Sherif MA, Dreßler K, Smeros P, Ngomo AN (2017) Radon - rapid discovery of topological relations. In: Proceedings of the Thirty-First AAAI conference on artificial intelligence, February 4–9, 2017, pp 175–181. AAAI Press, San Francisco, California, USA
15. Ngomo AN (2013) ORCHID - reduction-ratio-optimal computation of geo-spatial distances for link discovery. In: The Semantic Web - ISWC 2013–12th international semantic web conference, October 21–25, 2013, Proceedings, Part I, vol 8218. Lecture notes in computer science. Springer, Sydney, NSW, Australia, pp 395–410
16. Ahmed AF, Sherif MA, Ngomo AN (2018) On the effect of geometries simplification on geo-spatial link discovery. In: Proceedings of the 14th international conference on semantic systems, SEMANTiCS 2018, September 10–13, 2018. Procedia computer science, vol 137, pp 139–150. Elsevier, Vienna, Austria
17. Papadakis G, Mandilaras GM, Mamoulis N, Koubarakis M (2021) Progressive, holistic geospatial interlinking. In: WWW '21: The Web Conference 2021, April 19–23, 2021, pp 833–844. ACM / IW3C2, Virtual Event / Ljubljana, Slovenia
18. Papadakis G, Mandilaras G, Mamoulis N, Koubarakis M (2022) Static and dynamic progressive geospatial interlinking. ACM Trans Spatial Algorithms Syst 8(2):1–41
19. Siampou MD, Papadakis G, Mamoulis N, Koubarakis M (2023) Supervised scheduling for geospatial interlinking. In: Proceedings of the 31st ACM international conference on advances in geographic information systems, SIGSPATIAL 2023, November 13–16, 2023, pp 42–14212. ACM, Hamburg, Germany
20. Smeros P, Koubarakis M (2026) Discovering spatial and temporal links among RDF data. In: Proceedings of the workshop on linked data on the web, LDOW 2016. CEUR Workshop proceedings, vol. 1593. CEUR-WS.org, co-located with 25th International World Wide Web Conference (WWW 2016)
21. Fikioris G, Patroumpas K, Artikis A, Pitsikalis M, Paliouras G (2023) Optimizing vessel trajectory compression for maritime situational awareness. GeoInformatica 27(3):565–591
22. Pitsikalis M, Artikis A, Dreo R, Ray C, Camossi E, Jousselme A (2019) Composite event recognition for maritime monitoring. In: DEBS, pp 163–174. ACM, Darmstadt, Germany
23. Zhao B, Aa H, Nguyen TT, Nguyen QVH, Weidlich M (2021) EIRES: efficient integration of remote data in event stream processing. In: SIGMOD Conference, pp 2128–2141. ACM, Xi'an, Shaanxi, China

24. Khazael B, Vahidi-Asl M, Malazi HT (2023) Geospatial complex event processing in smart city applications. Simul Model Pract Theory 122:102675
25. Bruns R, Dunkel J, Seremet S (2023) Learning ship activity patterns in maritime data streams: Enhancing CEP rule learning by temporal and spatial relations and domain-specific functions. IEEE Trans Intell Transp Syst 24(10):11384–11395
26. Alevizos E, Artikis A, Paliouras G (2024) Complex event recognition with symbolic register transducers. Proceed VLDB Endowm 17(11):3165–3177
27. Alevizos E, Artikis A, Paliouras G (2022) Complex event forecasting with prediction suffix trees. VLDB J 31(1):157–180
28. Alevizos E, Artikis A, Paliouras G (2018) Wayeb: a tool for complex event forecasting. In: LPAR. EPiC Series in Computing, vol 57, pp 26–35. EasyChair, Awassa, Ethiopia
29. D'Antoni L, Veanes M (2017) The power of symbolic automata and transducers. In: CAV (1), vol 10426. Lecture notes in computer science. Springer, Heidelberg, Germany, pp 47–67
30. Nentwig M, Hartung M, Ngomo AN, Rahm E (2017) A survey of current link discovery frameworks. Semantic Web 8(3):419–436
31. Santipantakis GM, Vlachou A, Doulkeridis C, Artikis A, Kontopoulos I, Vouros GA (2018) A stream reasoning system for maritime monitoring. In: 25th International symposium on temporal representation and reasoning, TIME 2018, October 15–17, 2018. LIPIcs, vol 120, pp 20–12017. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Warsaw, Poland
32. Ray C, Dreo R, Camossi E, Jousselme A (2018) Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance. https://doi.org/10.5281/zenodo.1167595

Authors and Affiliations

Elias Alevizos[1,4] · Georgios M. Santipantakis[2] · Christos Doulkeridis[2] · Alexander Artikis[3,4]

✉ Elias Alevizos
alevizos.elias@iit.demokritos.gr

Georgios M. Santipantakis
gsant@unipi.gr

Christos Doulkeridis
cdoulk@unipi.gr

Alexander Artikis
a.artikis@unipi.gr

[1] The American College of Greece, Athens, Greece

[2] Department of Digital Systems, University of Piraeus, Piraeus, Greece

[3] Department of Maritime Studies, University of Piraeus, Piraeus, Greece

[4] NCSR, "Demokritos", Athens, Greece