Online Integration of Spatial Reasoning in Complex Event Recognition

Elias Alevizos[1,2], Georgios M. Santipantakis[2], Christos Doulkeridis[2] and Alexander Artikis[1,2]

[1]NCSR "Demokritos", Greece
[2]University of Piraeus, Greece

Abstract

Complex Event Recognition (CER) systems have the ability to process streams of events in real time 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 state and possibly previous events of the stream. CER systems are very efficient in tracking all the possible paths that a pattern may follow (since these are often non-deterministic) and report when a path is complete and a complex event must be reported. However, the conditions that need to be checked may be very demanding. For example, in the maritime monitoring domain, 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 engines which can evaluate this type of conditions very efficiently. Thus, we can integrate such a spatial reasoning engine within a CER engine in order to take advantage of both worlds: the CER engine can accommodate and process complex regular expressions and delegate the evaluation of expensive spatial tasks to the dedicated spatial reasoning engine. In this paper, we present an approach towards such an integration. We show how a CER engine can take advantage of a spatial reasoning engine. We describe two different communication schemes between the CER engine and the spatial reasoning engine (blocking and lazy) and explore when each one should be preferred.

Keywords

Finite Automata, Regular Expressions, Complex Event Recognition, Complex Event Processing, Symbolic Automata, Variable-order Markov Models

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, 2]. 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.

As an example, consider the case of maritime monitoring, which has gained considerable attention in the past yeas, both for economic and for environmental reasons [3, 4, 5, 6]. Monitoring vessel activity is critical for preventing risks and detecting suspicious or illegal behaviours, due to the Automatic Identification System (AIS)[1]. 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 [7]. 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 reasoning, called stLD [8], 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 as interlinked data.

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

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 engines and describe how they function in isolation. We then present how these two engines can cooperate and communicate with each other in Section 4. In Section 5 we present some experimental results and we conclude with Section 6.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2024 Joint Conference (March 25-28, 2024), Paestum, Italy
alevizos.elias@iit.demokritos.gr (E. Alevizos); gsant@unipi.gr (G. M. Santipantakis); cdoulk@unipi.gr (C. Doulkeridis); a.artikis@unipi.gr (A. Artikis)

[1]http://www.imo.org/OurWork/Safety/Navigation/Pages/AIS.aspx

2. Related Work

Link Discovery. Spatial link discovery or geospatial interlinking has been studied in [9, 10, 11]. 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 [12, 13]. In the same line of work, geospatial interlinking assisted by supervised learning has been studied in [14]. Notably, there is much less work for spatio-temporal link discovery [8, 15] which involves the movement of objects, and for real-time link discovery where the links need to be discovered over streaming input sources.

Complex Event Recognition. Complex Event Recognition systems have been studied extensively in the past decades (see [1, 2] 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 (possible 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. 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 [6, 16]. 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 [17]. 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.

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 engine whenever there is such a need. This engine 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 [18, 19]. 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 [20]. 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 [18]. 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.

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 $\mathcal{L}(R') = (\mathcal{L}(R))^c$ (^{c} stands for complement).*

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

Table 1
An example stream composed of five 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	...

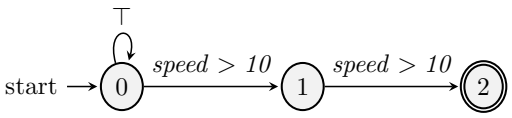


Figure 1: Streaming symbolic automaton created from the expression $R := (speed > 10) \cdot (speed > 10)$. The outgoing edges from state 0 are not mutually exclusive (they can be both triggered with the same event), thus rendering the automaton non-deterministic.

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 [20]). *A symbolic finite automaton (SFA) is a tuple $M = (Q, q^s, Q^f, \Delta)$, 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 $S_{m..k} \in \mathcal{L}(R)$.

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 Figure 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 $timestamp = 5$.

3.2. Spatiotemporal Link Discovery

In previous work, the stLD framework has been proposed to identify possibly complex spatial relations between data from streaming sources [8]. This is achieved using a blocking mechanism that groups data objects, so as to allow quick *filtering* of only a handful of candidate pairs that need to be examined during *refinement* for inclusion in the result. Typically, the temporal dimension has a significant role in spatiotemporal link discovery over streaming data sources. For instance, in the context of maritime situational awareness, it is useful to find vessels that come close to each other during a specific temporal interval, thus indicating a collision risk, rather than vessels that crossed the same area irrespective of when.

Since the spatial constraints need to be satisfied in conjunction with temporal constraint for discovering a spatiotemporal relation between entities, it follows that after a period of time some data can no longer lead to the discovery of new relations. Such data can be considered as obsolete and they can be eliminated by stLD, which supports a sliding window implementation. The sliding window has a fixed temporal duration defined at startup with respect to the temporal threshold and relations that need to be evaluated. The time window "slides" on the temporal dimension, as its starting and ending boundaries are being updated by the latest timestamp of the data received from the stream. Any data past the sliding window can be safely ignored, because they can no longer satisfy the temporal constraints of the relations. The elimination of obsolete data decreases the memory footprint and improves the overall performance, since fewer data need to be considered during link discovery. Within each window, the data objects are organized in memory using the blocking and optimization methods that are also available for the case of archival data sources.

Figure 2 illustrates the concept of blocking applied by stLD. A dataset reporting the position of vessels is denoted by plus symbol, and requests are illustrated by X mark. The center of the circles are the positions reported in the data or requests, while the range of circles are equal to the distance threshold used in proximity relations. In the illustrated example, the requests in cells (1,1), (1,2), (1,3) and (2,1) will be blocked in two cells. The requests in cells (1,2), (1,3), (2,1), (2,2) and (3,3) are satisfied by positions reported in the data set. Please notice that the requests in cells (1,1),(1,2),(1,3) and (2,1) are blocked in more than one cells. Similarly, entities in cells (1,3) and (3,2) are blocked in adjacent cells as well (since their extruded positions overlap with more than one cells). Any other geometry type (e.g., polygons) is also supported by stLD, however it is beyond the scope of this paper.

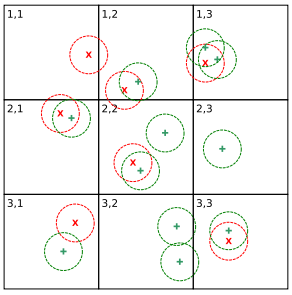


Figure 2: Space tiling for blocking a data set (denoted by plus symbol) and requests (denoted by X mark)

Some relations between entities may not be discovered by

considering only their reported positions, since the intermediate between consecutive positions are not reported in the data set. In this case, stLD constructs a line segment (i.e., a trajectory part) between consecutive positions, enabling the computation of intersecting points between trajectory parts and/or regions of interest (indicating the entry/exit points to a region of interest). Similarly, stLD employs bilinear interpolation when processing data retrieved from raster sources (e.g., bathymetry, weather forecast, etc). In this case the estimated value is computed from all the known neighboring positions of the request in the data set. In this work, stLD employs the bilinear interpolation to compute the sea depth at a requested position from a given binary (GRIB) file.

Finally, stLD applies an optimization technique called MaskLink [21] that can be applied on any type of geometry. MaskLink computes the area of each cell that is not occupied by any entity, the so-called "mask" of the cell that corresponds to empty space. More precisely, the mask is defined appropriately for a given relation under study, so as to ensure that any entity in the mask cannot be part the relation. Thus, any entity of the streaming data set that is found to be within the mask of a cell can be safely ignored from the refinement step of the link discovery process. In this way, MaskLink can restrict the number of candidates and improve the performance of link discovery.

4. CER powered with stLD

The stLD engine performs semantic spatio-temporal reasoning on raw stream data, whereas Wayeb performs real-time 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. Thus, Wayeb 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 system that can perform reasoning at various levels (see Figure 3).

4.1. Complexity of Event Recognition

We first have a closer look at how Wayeb works internally. The workflow is the following. 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 $Run(T_s, S_{..i})$, where $S_{..i}$ is the stream up to index i. Initially, before any input event has been consumed, the set of runs $Run(T_s, S_{..0})$ 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 $Run(T_s, S_{..k-1})$. After the event t_k, it produces $Run(T_s, S_{..k})$. This is achieved by evaluating t_k against every $\varrho \in Run(T_s, S_{..k-1})$. Each

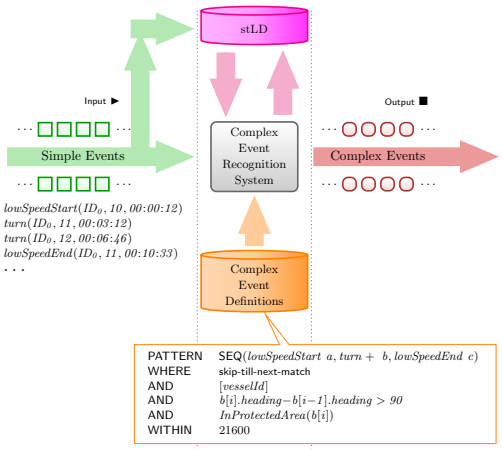


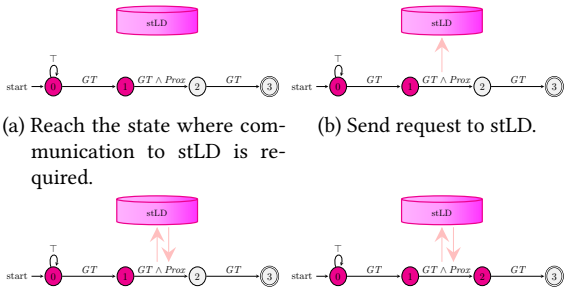
Figure 3: Overview of CER and stLD. The CER engine consumes a stream of annotated maritime events. Upon this stream, the system tries to detect instances of illegal fishing, defined as sequences of events where the vessel has low speed and executes one or more sudden turns. Additionally, we require that all turns are executed inside a protected area. The evaluation of the predicate of $InProtectedArea$ is delegated to the stLD engine which has a two-way communication with the CER engine. The stream is processed independently by the CER and stLD modules.

run $\varrho = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k]$ (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 $Run(T_s, S_{..k})$. If only one transition is triggered, then ϱ is updated, becoming $\varrho = [1, q_1] \xrightarrow{\delta_1} [2, q_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k] \xrightarrow{\delta_k} [k+1, q_{k+1}]$, with a new 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 $Run(T_s, S_{..k})$. 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 $Run(T_s, S_{..k})$. This process is repeated for the remaining runs of $Run(T_s, S_{..k-1})$.

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



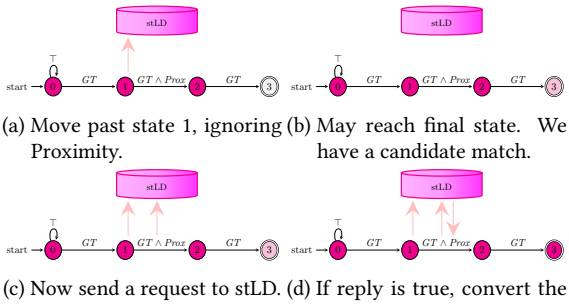
(a) Reach the state where communication to stLD is required.

(b) Send request to stLD.

(c) Wait until stLD replies.

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

Figure 4: Blocking scheme.



(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.

Figure 5: Lazy scheme.

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 engines is through a blocking scheme, as shown in Figure 4. *GT* stands for *Greater Than* and is a simple 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 engine. When an automaton run reaches a state, e.g., state 1 in Figure 4, whose outgoing transitions contain a spatial predicate, e.g., predicate *Prox* in Figure 4, then the CER engine sends a relevant request to stLD and blocks, waiting for a reply. 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).

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 Figure 5. 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 Figure 5 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 engines can then work in parallel. While the stLD engine 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 spatial 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 Figure 5 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 engine 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" (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 engine 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.

5. Empirical Evaluation

In this Section, we present relevant experimental results. We first test the stLD and CER modules independently, while working in standalone mode. Subsequently, we present results showing their combined performance.

5.1. stLD standalone

In this work, Wayeb depends on the spatiotemporal relations computed by stLD. We evaluate the performance of stLD on a 8-core QEMU Virtual CPU at 2.1GHz with 5GB dedicated memory to the Java VM. We use different distance and temporal thresholds for the proximity relation for each experiment, aiming to the computation of different number of relations. For each experiment stLD computes the sea depth at each reported position and the proximity relations w.r.t. the distance and temporal thresholds. We report the results for the set of experiments using the distance ($D\theta$) and temporal ($T\theta$) thresholds $d\theta(meters)$:{10, 100, 500, 1000}$\times T\theta(seconds)$:{60, 600, 1200} in Table 2, where the processing time and computed number of relations are reported in columns (per temporal threshold) for each distance threshold (in rows) evaluated. For example, the top left cells of the table, report that 122,518 proximity relations can be computed over the complete data set in 563,551 msec, using a distance threshold of 10 meters and temporal threshold of 60 seconds. The complete data covers a temporal interval of 4392 hours (183 days) and it is processed in 562 seconds (as an average of processing time in the results). A rough estimation of the processing time per relation is 0.128 msec per relation, which is computed from the average processing time over the average number of relations computed.

Table 2
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

5.2. CER standalone

An important feature of Wayeb is that it can detect patterns with very low latency. It can thus scale gracefully with increased loads. We first 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. The dataset that we used is publicly available, contains AIS kinematic messages from vessels sailing in the Atlantic Ocean around the port of Brest, France, and spans a period from 1 October 2015 to 31 March 2016. The total number of AIS messages is 18,648,556 from approximately 5,000 vessels [22, 23].

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 Figure 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 Figure 6 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.

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

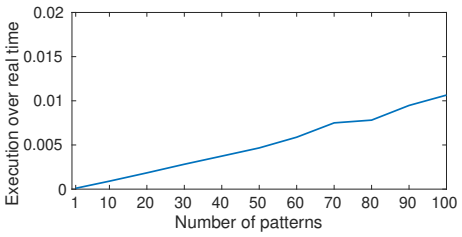


Figure 6: Ratio of execution over real time as a function of the number of patterns.

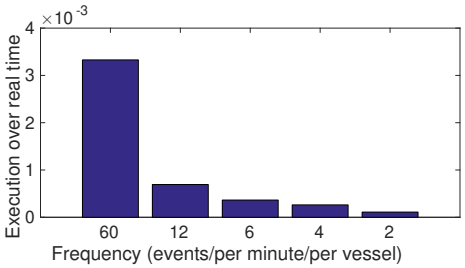


Figure 7: Ratio of execution over real time as a function of input event frequency.

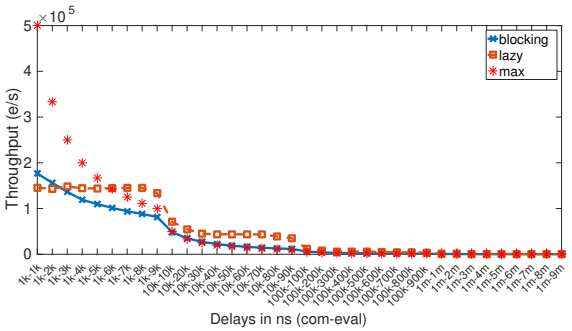


Figure 8: Throughput as a function of communication and evaluation delay. Each x point is a combination of a communication delay and an evaluation delay. For example, the x point with label $1k - 7k$ corresponds to an experiment where the communication delay was set to 1.000 ns and the evaluation delay to 7.000 ns.

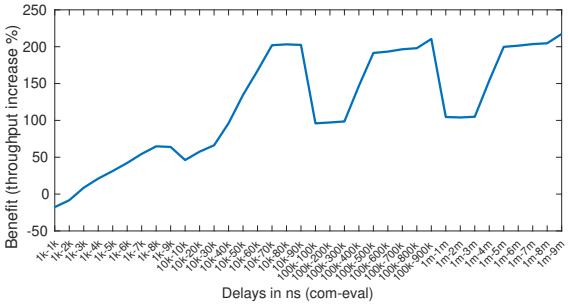


Figure 9: Throughput increase of lazy over blocking as a function of communication and evaluation delay. Negative values indicate that the blocking scheme is actually better.

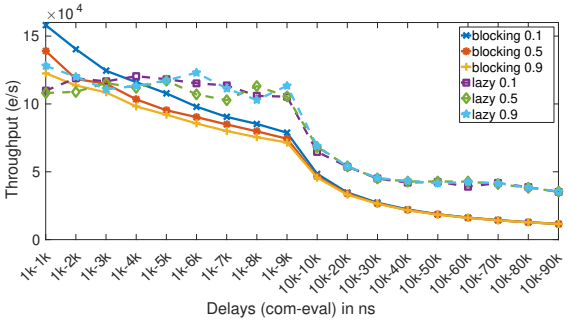


Figure 10: Throughput as a function of communication and evaluation delay for various values of satisfaction probability of the spatio-temproal predicate. We plot the performance of Wayeb for both blocking and lazy, for three different values of the satisfaction probability (0.1, 0.5, 0.9).

Figure 7. We observe that, again, Wayeb can process all workloads at a speed which is orders of magnitude greater than the event rate.

5.3. CER - stLD

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:

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

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 engine.

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 as TRUE. Figure 8 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 9 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 10 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 value of this satisfaction probability the fewer the time that the stLD will reply with TRUE to a request from the CEP engine.

In Figure 8, 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:

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

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 $WayebDelay$ is very small compared to the other two components (remember that $WayebDelay = 0$ 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 Figure 9 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.

Finally in Figure 10 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.

6. Discussion and Future 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, 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 communication 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.

Acknowledgment

This work was supported by the VesselAI project, under EU H2020 grant agreement No 957237.

References

[1] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, M. N. Garofalakis, Complex event recognition in the big data era: a survey, VLDB J. 29 (2020) 313–352.

[2] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Comput. Surv. 44 (2012) 15:1–15:62.

[3] B. Idiri, A. Napoli, The automatic identification system of maritime accident risk using rule-based reasoning, in: SoSE, IEEE, 2012, pp. 125–130.

[4] F. Terroso-Saenz, M. Valdés-Vela, A. F. Skarmeta-Gómez, A complex event processing approach to detect abnormal behaviours in the marine environment, Inf. Syst. Frontiers 18 (2016) 765–780.

[5] L. Snidaro, I. Visentini, K. Bryan, Fusing uncertain knowledge and evidence for maritime situational awareness via markov logic networks, Inf. Fusion 21 (2015) 159–172.

[6] K. Patroumpas, E. Alevizos, A. Artikis, M. Vodas, N. Pelekis, Y. Theodoridis, Online event recognition from moving vessel trajectories, GeoInformatica 21 (2017) 389–427.

[7] G. M. Santipantakis, A. Vlachou, C. Doulkeridis, A. Artikis, I. Kontopoulos, G. A. Vouros, A stream reasoning system for maritime monitoring, in: TIME, volume 120 of *LIPIcs*, 2018, pp. 20:1–20:17.

[8] G. M. Santipantakis, A. Glenis, C. Doulkeridis, A. Vlachou, G. A. Vouros, stLD: towards a spatio-temporal link discovery framework, in: SBD@SIGMOD, ACM, 2019, pp. 4:1–4:6.

[9] M. A. Sherif, K. Dreßler, P. Smeros, A. N. Ngomo, Radon - rapid discovery of topological relations, in: AAAI, AAAI Press, 2017, pp. 175–181.

[10] A. N. Ngomo, ORCHID - reduction-ratio-optimal computation of geo-spatial distances for link discovery, in: ISWC, volume 8218, Springer, 2013, pp. 395–410.

[11] A. F. Ahmed, M. A. Sherif, A. N. Ngomo, On the effect of geometries simplification on geo-spatial link discovery, in: SEMANTiCS, volume 137 of *Procedia Computer Science*, Elsevier, 2018, pp. 139–150.

[12] G. Papadakis, G. M. Mandilaras, N. Mamoulis, M. Koubarakis, Progressive, holistic geospatial interlinking, in: WWW, ACM / IW3C2, 2021, pp. 833–844.

[13] G. Papadakis, G. Mandilaras, N. Mamoulis, M. Koubarakis, Static and dynamic progressive geospatial interlinking, ACM Trans. Spatial Algorithms Syst. 8 (2022) 1–41.

[14] M. D. Siampou, G. Papadakis, N. Mamoulis, M. Koubarakis, Supervised scheduling for geospatial interlinking, in: SIGSPATIAL, ACM, 2023, pp. 42:1–42:12.

[15] P. Smeros, M. Koubarakis, Discovering spatial and temporal links among RDF data, in: LDOW@WWW, volume 1593 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016.

[16] M. Pitsikalis, A. Artikis, R. Dreo, C. Ray, E. Camossi, A. Jousselme, Composite event recognition for maritime monitoring, in: DEBS, ACM, 2019, pp. 163–174.

[17] B. Zhao, H. van der Aa, T. T. Nguyen, Q. V. H. Nguyen, M. Weidlich, EIRES: efficient integration of remote data in event stream processing, in: SIGMOD Conference, ACM, 2021, pp. 2128–2141.

[18] E. Alevizos, A. Artikis, G. Paliouras, Complex event forecasting with prediction suffix trees, VLDB J. 31 (2022) 157–180.

[19] E. Alevizos, A. Artikis, G. Paliouras, Wayeb: a tool for complex event forecasting, in: LPAR, volume 57 of *EPiC Series in Computing*, EasyChair, 2018, pp. 26–35.

[20] L. D'Antoni, M. Veanes, The power of symbolic automata and transducers, in: CAV (1), volume 10426 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 47–67.

[21] G. M. Santipantakis, C. Doulkeridis, G. A. Vouros, A. Vlachou, MaskLink: Efficient link discovery for spatial relations via masking areas, CoRR abs/1803.01135 (2018).

[22] C. Ray, R. Dreo, E. Camossi, A. Jousselme, Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, 10.5281/zenodo.1167595, 2018. URL: https://doi.org/10.5281/zenodo.1167595. doi:10.5281/zenodo.1167595.

[23] K. Bereta, K. Chatzikokolakis, D. Zissis, Maritime reporting systems, in: Guide to Maritime Informatics, Springer, 2021, pp. 3–30.