Noname manuscript No.
(will be inserted by the editor)

Online fleet monitoring with scalable event recognition and forecasting

Emmanouil Ntoulias · Elias Alevizos · Alexander Artikis · Charilaos Akasiadis · Athanasios Koumparos

Received: date / Accepted: date

Abstract Moving object monitoring is becoming essential for companies and organizations that need to manage thousands or even millions of commercial vehicles or vessels, detect dangerous situations (e.g., collisions or malfunctions) and optimize their behavior. It is a task that must be executed in real-time, reporting any such situations or opportunities as soon as they appear. Given the growing sizes of fleets worldwide, a monitoring system must be highly efficient and scalable. It is becoming an increasingly common requirement that such monitoring systems should be able to automatically detect complex situations, possibly involving multiple moving objects and requiring extensive background knowledge. Building a monitoring system that is both expressive and scalable is a significant challenge. Typically, the more expressive a system is, the less flexible it becomes in terms of its parallelization potential. We present a system that strikes a balance between expressiveness and scalability. Going beyond event detection, we also present an approach towards event forecasting. We show how event patterns may be given a probabilistic description so that our

E. Ntoulias
NCSR "Demokritos"
E-mail: manosntoulias@iit.demokritos.gr

E. Alevizos
National and Kapodistrian University of Athens, NCSR "Demokritos"
E-mail: ilalev@di.uoa.gr,alevizos.elias@iit.demokritos.gr

A. Artikis
University of Piraeus
NCSR "Demokritos"
E-mail: a.artikis@unipi.gr

C. Akasiadis
NCSR "Demokritos"
E-mail: cakasiadis@iit.demokritos.gr

A. Koumparos
Vodafone Innovus
E-mail: athkoumparos@gmail.com

system can forecast when a complex event is expected to occur. Our proposed system employs a formalism that allows analysts to define complex patterns in a user-friendly manner while maintaining unambiguous semantics and avoiding ad hoc constructs. At the same time, depending on the problem at hand, it can employ different parallelization strategies in order to address the issue of scalability. It can also employ different training strategies in order to fine-tune the probabilistic models constructed for event forecasting. Our experimental results show that our system can detect complex patterns over moving entities with minimal latency, even when the load on our system surpasses what is to be realistically expected in real-world scenarios.

Keywords Complex Event Processing · Big Data · Distributed Computing

1 Introduction

Commercial vehicle fleets constitute a major part of Europe's economy. There were approximately 37 million commercial vehicles in the European Union in 2015[1] and this number is growing every year with an increasing rate. Devices emitting spatial and operational information are installed on commercial vehicles. This information helps fleet management applications improve the management and planning of transportation services [48].

Consider another case of moving object monitoring, equally important from an economic and environmental point of view: maritime situational awareness [46,47,37]. The Automatic Identification System (AIS) [2] is used to track vessels at sea in real-time through data exchange with other ships nearby, coastal stations, or even satellites. Cargo ships of at least 300 gross tonnage and all passenger ships, regardless of size, are nowadays required to have AIS equipment installed and regularly emit AIS messages while sailing at sea. Currently, there are more than 500,000 vessels worldwide that can be tracked using AIS technology[3]. It is crucial, both for authorities and for maritime companies, to be able to track the behavior of ships at sea in order to avoid accidents and ensure that ships adhere to international regulations.

Streams of transient data emitted from vehicles or ships must be processed with minimal latency, if a monitoring system is to provide significant margins for action in case of critical situations. We therefore need to detect complex patterns of interest upon these streams in an online and highly efficient manner that can gracefully scale as the number of monitored entities increases. Besides kinematic data, it is also important to be able to take into account static (or "almost" static, with respect to the rate of the streaming data), background knowledge, such as weather data, point of interest (POI) information (like gas stations, ports, parking lots, police departments, NATURA areas where ships are not allowed to sail, etc [30]). This enhanced data stream

[1] http://www.acea.be/statistics/article/vehicles-in-use-europe-2017

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

[3] https://www.vesselfinder.com

produces valuable opportunities for the detection of complex events. One can identify certain routes a vehicle or a ship is taking, malfunctions in the GPS tracker or the AIS transponder, cases of illegal shipping in protected areas or possible collisions between ships moving dangerously close to each other, to name but a few of the possible patterns which could be of interest to analysts. Besides detecting patterns of interest, another important functionality is that of forecasting whether such a pattern might occur in the future before it actually occurs. Especially in the domain of moving object monitoring and fleet management, such a functionality could allow analysts to have significant margins of action by responding to alarms indicating the possible occurrence of critical situations. Collision avoidance is an obvious example where forecasting is significantly more important than simple detection, but other patterns are also useful to forecast, such as predicting whether a ship is about to enter an anchorage area or a port so that the traffic around a port may be more efficiently managed.

As a solution to the problem of monitoring of moving objects, we present a Complex Event Processing and Forecasting system that aims to improve the operating efficiency of a commercial fleet. It operates online with enriched data in a streaming environment. Our contributions are the following:

- We present a Complex Event Processing and Forecasting (CEP/F) system based on symbolic automata and Markov models which allows analysts to define complex patterns in a user-friendly language. Our proposed language has formal semantics, while being able to also take into account background knowledge.
- We define a series of realistic complex patterns that identify routes and malfunctions of vehicles and detect critical situations for vessels at sea.
- We present and compare various parallel processing techniques and discuss their applicability. We use them to implement a parallel system for online recognition, training and forecasting.
- We augment the parallel training/forecasting implementation with basic adaptation capabilities that change on-the-fly characteristics of the trained model.
- We test our approach using large, real-world, heterogeneous data streams from diverse application domains, showing that we can achieve real-time performance even in cases of significantly increased load, beyond the current demand levels.

This paper extends the work that we have already presented in [35]. It presents a more comprehensive and detailed description of our engine. It also describes its forecasting capabilities (see also [13]). We enrich the experiments presented in [35] to include results on forecasting and we also present a first step towards adaptive forecasting where the engine needs to adapt its forecasting model(s) on-the-fly.

The remainder of this paper is organized as follows. Section 2 discusses related work, while Section 3 describes our CEP/F engine. In Section 4 the parallel version of our engine is presented. Section 5 summarizes the datasets of

vehicle and vessel traces and defines the recognition patterns. It also presents our empirical evaluation. Finally, we conclude the paper in section 6 and describe our future work.

2 Related Work

Complex event recognition systems accept as input a stream of time-stamped, "simple, derived events" (SDEs). These SDEs are the result of applying a simple transformation to some other event (e.g., a measurement from a sensor). By processing them, a CEP engine can recognize complex events (CEs), i.e. collections of SDEs satisfying some pattern. There are multiple CEP systems proposed in the literature during the last 15 years, falling under various classes [22,14,27]. Automata-based systems constitute the most common category. They compile patterns (definitions of complex events) into finite state automata, which are then used to consume streams of simple events and report matches whenever an automaton reaches a final state. Examples of such systems may be found in [24,50,45,5,12]. Another important class of CEP systems are the logic-based ones. In this case, patterns are defined as rules, with a head and a body defining the conditions which, if satisfied, lead to the detection of a CE. A typical example of a logic-based system may be found in [15]. Finally, there are some tree-based systems, such as [33,32], which are attractive because they are amenable to various optimization techniques.

For efficient processing on big data streams, distributed architectures need to be employed [27]. Big data platforms, such as Apache Spark and Storm, have been used to embed CEP engines into their operators. Both platforms have incorporated Sidhi [7,9] and Esper [3,4] as their embedded engines. Flink [1], on the other hand, provides support for CEP with the FlinkCEP built-in library [5]. Besides using these Big Data platforms, numerous other parallelization techniques have been proposed in the literature that can achieve a more fine-grained control over how the processing load is distributed among workers. Pattern-based parallelization is the most obvious solution, where the patterns are distributed among the processing units [21]. One disadvantage of this parallelization scheme is that events have to be replicated to multiple processing units, since a new input event may need to be consumed by more than one pattern. Moreover, the parallelization level is necessarily limited by the number of patterns (for a single pattern, this method offers no benefits). Operator-based parallelization constitutes another approach, where the CEP operators are assigned to different processing units [45,16]. This allows for multi-pattern optimizations and avoids the data replication issue of the previous technique. On the other hand, the parallelization level is again limited, this time by the number of operators present in the pattern (which is closely related to the number of automaton states in automata-based CEP systems). Finally, in data-parallelization schemes, events are split among multiple instances of the same pattern [29]. For example, a pattern trying to detect violations of speed limits must be applied to all the monitored vehicles and thus the input

stream may be partitioned according to the id of the vehicles. The advantage of this method is that it can scale well with the input event rate. It is, however, not always obvious how an input stream should be partitioned, while avoiding data replication.

Forecasting has not received much attention in the field of CEP. Some conceptual proposals have acknowledged the need for CEF though [26, 25, 19]. The first concrete attempt at Complex Event Forecasting (CEF) was presented in [34], where a variant of regular expressions and automata was used to define complex event patterns, along with Markov chains. Each automaton state was mapped to a Markov chain state. Symbolic automata and Markov chains were again used in [11, 12]. The problem with these approaches is that they are essentially unable to encode higher-order dependencies, since high-order Markov chains may lead to a combinatorial explosion of the number of states. In [36], complex events were defined through transitions systems and Hidden Markov Models (HMM) were used to construct a probabilistic model. The observable variable of the HMM corresponded to the states of the transition system. HMMs are in general more powerful than Markov chains, but, in practice, the may be hard to train ([17, 10]) and require elaborate domain modeling, since mapping a pattern to a HMM is not straightforward. In contrast, our approach constructs seamlessly a probabilistic model from a given CE pattern (declaratively defined). Knowledge graphs were used in in [31] to encode events and their timing relationships. Stochastic gradient descent was employed to learn the weights of the graph's edges that determine how important an event is with respect to another target event. However, this approach falls in the category of input event forecasting, as it does not target complex events.

3 Automata-based Complex Event Processing and Forecasting

We begin by first presenting our framework for CEP. It is based on Wayeb, a Complex Event Processing and Forecasting engine which employs symbolic automata as its computational model and Markov models as a probabilistic framework [12, 11]. The rationale behind our choice of Wayeb is that, contrary to other automata-based CEP engines, it has clear, compositional semantics due to the fact that symbolic automata have nice closure properties [23]. At the same time, it is expressive enough to support most of the common CEP operators [27], while remaining amenable to the standard parallelization solutions. In this paper, we extend Wayeb's language in order to support more expressive patterns.

3.1 Complex Event Processing

Symbolic automata constitute a variation of classical automata, with the main difference being that their transitions, instead of being labeled with a symbol

from an alphabet, are equipped with formulas from Boolean algebra [23]. A symbolic automaton consumes strings and, after every new element, applies the predicates of its current state's outgoing transitions to that element. If a predicate evaluates to TRUE then the corresponding transition is triggered and the automaton moves to that transition's target state. A Boolean algebra is defined as follows:

Definition 1 (Boolean algebra [23]) A Boolean algebra is a tuple ($\mathcal{D}$, Ψ, $[\![\_]\!]$, $\bot$, $\top$, $\vee$, $\wedge$, $\neg$) where $\mathcal{D}$ is a set of domain elements; Ψ is a set of predicates closed under the Boolean connectives; $\bot, \top \in \Psi$; the component $[\![\_]\!] : \Psi \to 2^{\mathcal{D}}$ is a denotation function such that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathcal{D}$ and $\forall \phi, \psi \in \Psi$:

- $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$;
- $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$;
- $[\![\neg\phi]\!] = \mathcal{D} \setminus [\![\phi]\!]$.

Elements of $\mathcal{D}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathcal{L}$ constructed from elements of $\mathcal{D}$ ($\mathcal{L} \subseteq \mathcal{D}^*$, where * denotes Kleene-star) is called a language over $\mathcal{D}$.

Wayeb uses symbolic regular expressions to define patterns and to represent a class of languages over $\mathcal{D}$ [13]. Wayeb's standard operators are those of the classical regular expressions, i.e., concatenation, disjunction and Kleene-star. We extend Wayeb to include various extra CEP operators: that of negation and those of different selection policies (see [27] for a discussion of selection policies). Symbolic regular expressions are defined as follows:

Definition 2 (Symbolic regular expression) A Wayeb symbolic regular expression (*SRE*) over a Boolean algebra ($\mathcal{D}$, Ψ, $[\![\_]\!]$, $\bot$, $\top$, $\vee$, $\wedge$, $\neg$) is recursively defined as follows:

- If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathcal{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the subset of $\mathcal{D}$ 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.
- Bounded iteration: If R is a symbolic regular expression, then $R' := R^{x+}$ is a symbolic regular expression, with $R^{x+} = \overbrace{R \cdot \cdots \cdot R}^{x \ times} \cdot R^*$.
- 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 $R' := R_1 \cdot !(\top^* \cdot R_2 \cdot \top^*) \cdot R_2 \cdots !(\top^* \cdot R_n \cdot \top^*) \cdot R_n$.

A Wayeb expression without a selection policy implicitly follows the strict-contiguity policy, i.e., the SDEs involved in a match of a pattern should occur contiguously in the input stream. The other two selection policies relax the strict requirement of contiguity (see [27] for details). For example, if we use the skip-till-any-match policy as defined above, then we allow any number of "irrelevant" events to occur between matches of the sub-expressions $R_1, R_2, \cdots, R_n$. Note that all these operators, even those of selection policies, may be arbitrarily used and nested in an expression, without any limitations. This is in contrast to other CEP systems where nested operations may be prohibited.

Wayeb patterns are defined as symbolic regular expressions which are subsequently compiled into symbolic automata. The definition for a symbolic automaton is the following:

Definition 3 (Symbolic finite automaton [23]) A symbolic finite automaton (*SFA*) is a tuple $M = (\mathcal{A}, Q, q^s, F, \Delta)$, where $\mathcal{A}$ is an effective Boolean algebra; 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; $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of transitions.

A string $w = a_1 a_2 \cdots a_k$ is accepted by a *SFA* M iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q^s$ and $q_k \in Q^f$. The set of strings accepted by M is the language of M, denoted by $\mathcal{L}(M)$. It can be proven that every symbolic regular expression can be translated to an equivalent (i.e., with the same language) symbolic automaton [23].

We are now in a position to precisely define the meaning of "complex events". Input events come in the form of tuples with both numerical and categorical values. These tuples constitute the set of domain elements $\mathcal{D}$. A stream S is an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a tuple ($t_i \in \mathcal{D}$). Our goal is to report the indices i at which a CE 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 $S_{m..k} \in \mathcal{L}(R)$. If we attempted to detect CEs, as defined above, by directly compiling an expression R to an automaton, we would fail. Consider, for example, the (classical) regular expression $R := a \cdot b$ and the (classical) stream/string $S = a, b, c, a, b, c$. If we compile R to a (classical) automaton and feed S to it, then the automaton would reach its final state after reading the second element t_2 of the string. However, it would then never reach its final state again. We would like our automaton to reach its final state every time it encounters a, b as a suffix, e.g., again after reading t_5 of S. We can achieve this with a simple trick. Instead of using R, we first convert it to $R_s = \top^* \cdot R$. Using R_s we can detect CEs of R while consuming a stream S, since a stream segment $S_{m..k}$ is

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

vehicle id	78986	78986	78986	78986	78986	...
speed	85	93	99	104	111	...
timestamp	1	2	3	4	5	...

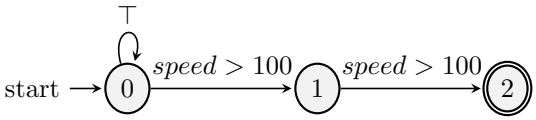


Fig. 1: Streaming symbolic automaton created from the expression $R := (speed > 100) \cdot (speed > 100)$.

recognized by R iff the prefix $S_{1..k}$ is recognized by R_s. The prefix $\top^*$ lets us skip any number of events from the stream and start recognition at any index $m, 1 \leq m \leq k$.

As an example, consider the domain of vehicle monitoring. An analyst could use the Wayeb language to define the pattern $R := (speed > 100) \cdot (speed > 100)$ in order to detect speed violations on roads where the maximum allowed speed is 100 km/h. This pattern detects two consecutive events where the speed exceeds the threshold in order to avoid cases where a vehicle momentarily exceeds the threshold, possibly due to some measurement error. 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 $timestamp = 5$.

Note that our engine has been designed to be as generic as possible. It can be used in a domain such as moving object monitoring, where the geospatial component is dominant, but it can also be used in other domains which fall completely outside this field, such as credit card fraud management. The experimental results that we present in Section 5 have thus been obtained without employing any domain-specific optimizations. Such optimizations are of course conceivable and could presumably yield significant advantages. For example, evaluating whether a given position lies within a given polygon can be performed efficiently by partitioning geographical areas through the use of a grid. We intend to pursue this line of work in the future, but for now we restrict our attention to the problem of boosting efficiency through parallelization techniques in the most generic and widely applicable manner. We would also like to mention that our queries target individual objects and not groups of objects (as in [43,44]). Our engine does not in principle preclude group queries.

We will explore in the future whether modifications in the architecture of the engine are required in order to accommodate such queries.

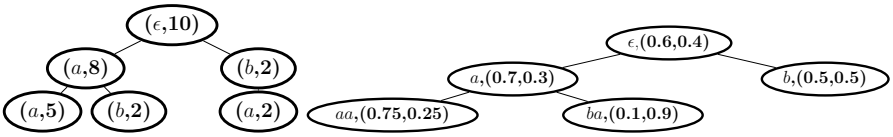
3.2 Complex Event Forecasting

We now show how we can use the framework of symbolic automata to perform Complex Event Forecasting (for more details, see [13]). The main idea behind our forecasting method is the following: Given a pattern R in the form of a symbolic regular expression, we first construct an automaton. In order to perform event forecasting, we translate the automaton to an equivalent deterministic symbolic automaton. This deterministic automaton can then be used to learn a probabilistic model, typically a Markov model, that encodes dependencies among the events in an input stream. Note that deterministic automata are important because they allow us to produce a stream of "symbols" from the initial stream of events. By using deterministic automata, we can map each input event to a single symbol and then use this derived stream of symbols to learn a Markov model. By definition, if we are at any time in a given state of a deterministic automaton, only one of the outgoing transitions may be triggered. Thus, if we assign a unique symbol to each transition, then we can map a stream of events to a stream of symbols: the symbols corresponding to the transitions that were triggered after reading each input event. The probabilistic model is learned from a portion of the input stream which acts as a training dataset. It is then used to derive forecasts about the expected occurrence of the complex event encoded by the automaton. After learning a model, we need to estimate the so-called *waiting-time distributions* for each state of our automaton. These distributions let us know the probability of reaching a final state from any other automaton state in k events from now. These distributions are then used to estimate forecasts, which generally have the form of an interval within which a complex event has a high probability of occurring.

We propose the use of a variable-order Markov model (VMM) [17, 42, 41, 20, 49]. Compared to full-order Markov models, VMMs allow us to increase their order m (how many events they can remember) to higher values and thus capture longer-term dependencies, which can lead to a better accuracy. We use Prediction Suffix Trees, as described in [42, 41], as our VMM of choice. The formal definition of a PST is the following:

Definition 4 (Prediction Suffix Tree [42]) Let Σ be an alphabet. A PST T over Σ is a tree whose edges are labeled by symbols $\sigma \in \Sigma$ and each internal node has exactly one edge for every $\sigma \in \Sigma$ (hence, the degree is $\mid \Sigma \mid$). Each node is labeled by a pair (s, γ_s), where s is the string associated with the walk starting from that node and ending at the root, and $\gamma_s : \Sigma \rightarrow [0, 1]$ is the next symbol probability function related with s. For every string s labeling a node, $\sum_{\sigma \in \Sigma} \gamma_s(\sigma) = 1$. The depth of the tree is its order m.

Figure 2b shows an example of a Prediction Suffix Tree of order $m = 2$. According to this tree, if the last symbol that we have encountered in a stream is



(a) Example of a Counter Suffix Tree with $m = 2$ and $S = aaabaabaaa$.

(b) Example of a Prediction Suffix Tree T for $\Sigma = \{a, b\}$ and $m = 2$. Each node contains the label and the next symbol probability distribution for a and b.

Fig. 2: Examples of Counter and Prediction Suffix Trees.

a and we ignore any other symbols that may have preceded it, then the probability of the next input symbol being again a is 0.7. However, we can obtain a better estimate of the next symbol probability by extending the context and looking one more symbol deeper into the past. Thus, if the last two symbols encountered are b, a, then the probability of seeing a again is very different (0.1). On the other hand, if the last symbol encountered is b, the next symbol probability distribution is $(0.5, 0.5)$ and, since the node $b, (0.5, 0.5)$ has not been expanded, this implies that its children would have the same distribution if they had been created. Therefore, the past does not affect the prediction and will not be used. Note that a Prediction Suffix Tree whose leaves are all of equal depth m corresponds to a full-order Markov model of order m, as its paths from the root to the leaves correspond to every possible context of length m.

Our goal is to incrementally learn a Prediction Suffix tree $\hat{T}$ by adding new nodes only when it is necessary. First, we need to derive the initial empirical conditional distributions about the various symbols (e.g., $P(a \mid b)$, the probability of seeing a after b). In [42], it is assumed that the empirical probabilities of symbols given various contexts are available. The suggestion in [42] is that these empirical probabilities can be calculated either by repeatedly scanning the training stream or by using a more time-efficient algorithm that keeps pointers to all occurrences of a given context in the stream. We opt for a variant of the latter choice. We basically need to count the number of occurrences of the various candidate strings s in $S_{1..k}$. In order to keep track of these counters, we can use a tree data structure which allows to scan the training stream only once. We call this structure a *Counter Suffix Tree*. Each node in a Counter Suffix Tree is a tuple (σ, c) where σ is a symbol from the alphabet (or ϵ only for the root node) and c a counter. By following a path from the root to a node, we get a string $s = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$, where $\sigma_0 = \epsilon$ corresponds to the root node. The property maintained as a Counter Suffix Tree is built from a stream $S_{1..k}$ is that the counter of the node σ_n that is reached with s gives us the number of occurrences of the string $\sigma_n \cdot \sigma_{n-1} \cdots \sigma_1$ (the reversed version of s) in $S_{1..k}$. As an example, see Figure 2a, which depicts the Counter Suffix Tree of maximum depth 2 for the stream $S = aaabaabaaa$. If we want to retrieve the number of occurrences of the string $b \cdot a$ in S, we follow the left

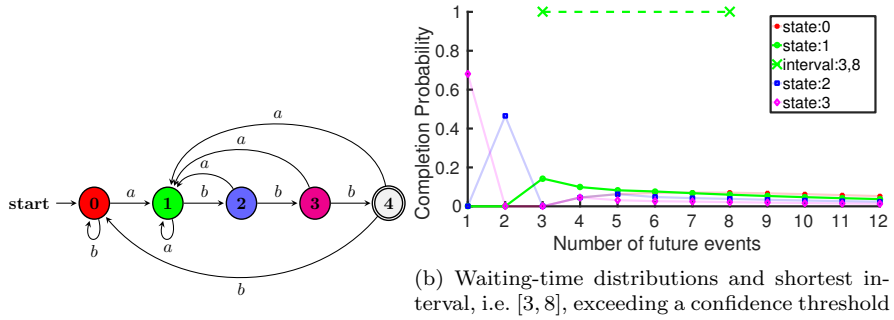
child $(a, 7)$ of the root and then the right child of this. We thus reach $(b, 2)$ and indeed $b \cdot a$ occurs twice in S.

After we have a Counter Suffix Tree available, we can then try to learn a Prediction Suffix Tree. The learning algorithm in [42] starts with a tree having only a single node, corresponding to the empty string ϵ. Then, it decides whether to add a new context/node s by checking two conditions:

- First, there must exist $\sigma \in \Sigma$ such that $\hat{P}(\sigma \mid s) > \theta_1$ must hold, i.e., σ must appear "often enough" after the suffix s;
- Second, $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} > \theta_2$ (or $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} < \frac{1}{\theta_2}$) must hold, i.e., it is "meaningful enough" to expand to s because there is a significant difference in the conditional probability of σ given s with respect to the same probability given the shorter context $suffix(s)$, where $suffix(s)$ is the longest suffix of s that is different from s.

For example, consider node a in Figure 2b and assume that we are at a stage of the learning process where we have not yet added its children, aa and ba. We now want to check whether it is meaningful to add ba as a node. Assuming that the first condition is satisfied, we can then check the ratio $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} = \frac{\hat{P}(a|ba)}{\hat{P}(a|a)} = \frac{0.1}{0.7} \approx 0.14$. If $\theta_2 = 1.05$, then $\frac{1}{\theta_2} \approx 0.95$ and the condition is satisfied, leading to the addition of node ba to the tree. It can be proven that the hypothesis tree $\hat{T}$ constructed via the above procedure, is an ϵ-good hypothesis (where ϵ is an approximation parameter) with respect to the source that generates the training dataset, i.e., the KL-divergence between the distributions of $\hat{T}$ and the original source is bounded above by ϵ. For more details, see [42].

We can use a Prediction Suffix Tree T to calculate the so-called waiting-time distribution for every state q of an automaton A, i.e., the distribution of the index n, given by the waiting-time variable $W_q = inf\{n : Y_0, Y_1, ..., Y_n\}$, where $Y_0 = q$, $Y_i \in A.Q \backslash A.Q_f$ for $i \neq n$ and $Y_n \in A.Q_f$. Given an automaton A and its Prediction Suffix Tree T, we can estimate the probability for A to reach for the first time one of its final states in the following manner. As the system processes events from the input stream, besides feeding them to A, it also stores them in a buffer that holds the m most recent events, where m is equal to the maximum order of the Prediction Suffix Tree T. After updating the buffer with a new event, the system traverses T according to the contents of the buffer and arrives at a leaf l of T. The probability of any future sequence of events can be estimated with the use of the probability distribution at l. In other words, if $S_{1..k} = \cdots, t_{k-1}, t_k$ is the stream seen thus far, then the next symbol probability for t_{k+1}, i.e., $P(t_{k+1} \mid t_{k-m+1}, \cdots, t_k)$, can be directly retrieved from the distribution of the leaf l. If we want to look further into the future, e.g., into t_{k+2}, we can repeat the same process as necessary. Namely, if we fix t_{k+1}, then the probability for t_{k+2}, $P(t_{k+2} \mid t_{k-m+2}, \cdots, t_{k+1})$, can be retrieved from T, by retrieving the leaf $l^{'}$ reached with $t_{k+1}, \cdots, t_{k-m+2}$. In this manner, we can estimate the probability of any future sequence of events. Consequently, we can also estimate the probability of any future sequence of states of A, since we can simply feed these future event sequences



(a) Deterministic automaton.

(b) Waiting-time distributions and shortest interval, i.e. $[3, 8]$, exceeding a confidence threshold $\theta_{fc} = 50\%$ for state 1.

Fig. 3: Automaton and waiting-time distributions for $R = a \cdot b \cdot b \cdot b$, $\Sigma = \{a, b\}$.

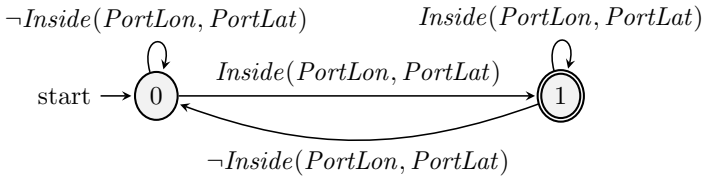


Fig. 4: Deterministic symbolic automaton created from Pattern (1).

to A and let it perform "forward" recognition with these projected events. In other words, we can let A "generate" a sequence of future states, based on the sequence of projected events, in order to determine when A will reach a final state. Finally, since we can estimate the probability for any future sequence of states of A, we can use the definition of the waiting-time variable $(W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in A.Q \backslash A.Q_f, Y_n \in A.Q_f\})$ to calculate the waiting-time distributions.

Figure 3 shows an example of an automaton (its exact nature is not important, as long as it can also be described as a Markov chain), along with the waiting-time distributions for its non-final states. For this example, if the automaton is in state 2, then the probability of reaching the final state 4 for the first time in 2 transitions is $\approx 50\%$. However, it is 0% for 3 transitions, as the automaton has no path of length 3 from state 2 to state 4.

In order to ground the above discussion in the domain of moving object monitoring, we will provide here a more concrete example. Assume that we are interested in the following very simple pattern:

$$InsidePort := Inside(PortLon, PortLat) \tag{1}$$

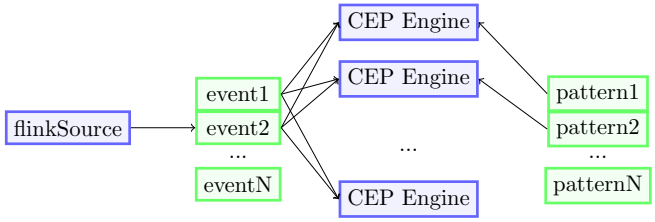
This pattern consists of a single predicate that evaluates to TRUE whenever a ship is within the boundaries of a given port (the exact details of how this predicate is evaluated are not relevant here). If we wanted to detect only entrances to the port (and not exits), we would need to enhance this pattern by requiring that complex events are detected only when the ship was first

outside the port and then, with the next input event, inside it. We could use the concatenation operator to define this pattern, along with logical negation, as follows:

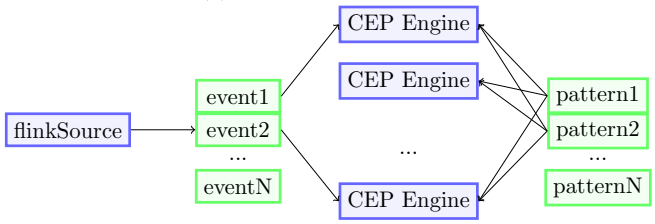
$$InsidePort := \neg Inside(PortLon, PortLat) \cdot Inside(PortLon, PortLat) \quad (2)$$

For reasons of simplicity, we will base the following discussion on Pattern (1). The deterministic symbolic automaton corresponding to this pattern is shown in Figure 4. This automaton has two "symbols": a corresponding to $Inside(PortLon, PortLat)$ and b for its negation $\neg Inside(PortLon, PortLat)$. After constructing the automaton, we can use a training dataset to learn the relevant Counter and Prediction Suffix Trees, e.g., as those shown in Figure 2. This is achieved by converting the training stream of position signals to a stream of symbols composed only of as and bs. As we have explained above, each position signal corresponds exactly to one symbols, either a (meaning that the position is inside the port) or b (meaning that the position is outside the port). The learnt Prediction Suffix Tree of Figure 2b should be interpreted in this example as follows: the left-most node $aa, (0.75, 0.25)$ implies that the probability of the ship being inside the port (symbol a) is 0.75, given that it was already inside the port for the last two positions. Similarly for the other nodes. With the help of the Prediction Suffix Tree we can then calculate the waiting-time distributions for the automaton of Figure 4.

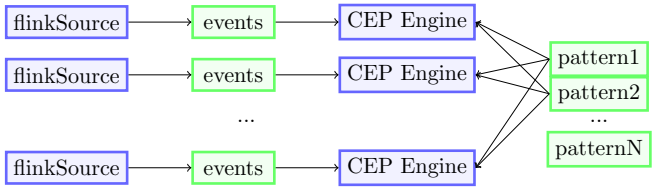
We can use the waiting-time distributions to produce various kinds of forecasts. In the simplest case, we can select the future point with the highest probability and return this point as a forecast. We call this type of forecasting *REGRESSION-ARGMAX*. Alternatively, we may want to know how likely it is that a CE will occur within the next w input events. For this, we can sum the probabilities of the first w points of a distribution and if this sum exceeds a given threshold we emit a "positive" forecast (meaning that a CE is indeed expected to occur); otherwise a "negative" (no CE is expected) forecast is emitted. We call this type of forecasting *CLASSIFICATION-NEXTW*. These kinds of forecasts are easy to compute. There is another kind of useful forecasts, which are however more computationally demanding. Given that we are in a state q, we may want to forecast whether the automaton, with confidence at least θ_{fc}, will have reached its final state(s) in n transitions, where n belongs to a future interval $I = [start, end]$. The confidence threshold θ_{fc} is a parameter set by the user. The forecasting objective is to select the shortest possible interval I that satisfies θ_{fc}. Figure 3b shows the forecast interval produced for state 1 of the automaton of Figure 3a, with $\theta_{fc} = 50\%$. We call this third type of forecasting *REGRESSION-INTERVAL*. In this paper and the empirical results that we present in what follows, we focus on *CLASSIFICATION-NEXTW*. The reason is that this type of forecasting can be more comprehensively evaluated in the context of CEP. With *CLASSIFICATION-NEXTW* we can evaluate how our solution performs both against positives (i.e., when the complex event does indeed occur within the specified window) and against negatives (i.e., when no complex event occurs). Given that complex events are generally rare and thus negatives are frequent,



(a) Pattern-based parallelization.



(b) Partition-based parallelization.



(c) Special case of Partition-Based parallelization when one-to-one relation between sources and CEP engines exists.

Fig. 5: Parallel schemes used with Wayeb.

the performance of a CEF method against negatives is crucial. Measuring this performance with regression forecasting (either *REGRESSION-INTERVAL* or *REGRESSION-ARGMAX*) is, however, not possible, since these types of forecasting always assume that a complex event will happen and they then measure the difference between the predicted timestamp (or interval) and the actual one. In the future, we intend to investigate if and how regression forecasting can provide meaningful ways to evaluate the performance of a CEF engine.

4 Scalable CEP/F Over Multiple Trajectories

We now discuss how various parallelization schemes may be applied to our CEP engine. For this purpose, we leverage a popular streaming platform, Apache Flink [18, 1]. Flink is a distributed processing engine for stateful computations over unbounded and bounded data streams. It is designed to run in cluster environments and perform computations at in-memory speed. In this paper, we focus on two parallelization techniques: pattern-based and partition-based parallelization [27]. We currently exclude state-based parallelization, since, as

explained above (Section 2), its parallelization level is limited by the number of automaton states, which is typically quite low (it is often a single-digit number). We do intend, however, to examine in the future how it could be combined with pattern- or partition-based parallelization to provide them with an extra performance boost.

4.1 Scalable Event Processing

In pattern-based parallelization, each available CEP engine receives a unique subset of the patterns and performs recognition for these patterns only, with these subsets being (almost) equal in size (see Figure 5a). On the other hand, the stream is broadcast to all parallel instances of any downstream operators. This is a significant (yet unavoidable) drawback of pattern-based parallelization, since each worker has a subset of the patterns while each pattern may need to process the whole stream. Note that each blue rectangle in Figure 5a represents a single thread. This means that in this architecture we have one thread for the source plus as many threads as the parallelism of the CEP operator.

In partition-based parallelization the opposite happens. Every CEP engine is initialized with all patterns, but the stream is not broadcast (see Figure 5b). A partitioning function is used to decide where each new input event should be forwarded. This function takes as input any attribute of the event (we use the id of a vehicle or vessel) and, by performing hashing, it outputs which parallel instance of the next operator the event will go to. As with pattern-based parallelization, we have one thread for the source plus as many threads as the parallelism of the CEP operator.

Besides Flink, we also use the Apache Kafka messaging platform to connect our stream sources to Wayeb instances [2]. Kafka provides various ways to consume streams. So far, we have focused on linear streams, i.e., events are assumed to be totally ordered and arrive at our system sequentially one after another. With Kafka, however, there is the option of using parallel input streams. A Kafka input topic can have multiple partitions and each partition can be consumed in parallel by a different consumer. In this case the input stream is already partitioned on some attribute of the events.

Through this Kafka functionality, a variant of partition-based parallelization becomes possible, where both the input source and the recognition engine work in parallel (see Figure 5c). If the parallelism of the input source (i.e., the number of partitions of the topic is the same as that of the recognition operator (i.e., number of CEP engines), then we can simply attach each source instance to a CEP engine instance without further re-partitioning on our end. When, however, the parallelisms are different, further re-partitioning is performed by partitioning each source the same way we partitioned the single threaded source. Unlike the previous architectures, each pair of a source and a CEP operator parallel instances belong in a single thread. This is attributed

to operator chaining [8], a Flink mechanism that chains operators of the same parallelism in a single thread for better performance.

Similarly to partition-based parallelization, we can have multiple sources for pattern-based parallelism as well. Messages in Kafka are still partitioned by a desired attribute, albeit we always have to perform the broadcasting step for each source. Hence, we don't have a variant of pattern parallelization, rather we use it only as a way to test Wayeb with streams of higher input rate.

4.2 Scalable Training and Forecasting

We present the parallel version of Wayeb, enhanced with adaptation capabilities. Figure 6 depicts the Flink operators comprising the module. Events are first loaded into a Kafka topic and then forwarded to the training and the forecasting operators. Note that training and forecasting (i.e., the emission of forecasts) are two separate sub-processes within this module. Training involves the use of a dataset to estimate suffix trees, distributions and forecast intervals, as described in Section 3.2. This step can be performed offline on a training dataset and the constructed model can then be used for forecasting on multiple testing datasets, assuming that the testing datasets have the same statistical properties as the training one. In what follows, we will show how this step may be performed online and thus how we can avoid relying on the stationarity assumption. Forecasting, on the other hand, involves the emission of the forecast intervals from each state. As soon as an automaton reaches a given state, it can emit the interval of this state as a forecast, if the confidence for that interval is above a user-provided threshold. This step is always performed online.

The training operator is a co-flatmap function (i.e., a function that has two different input streams and for each input message of an input stream produces zero, one or multiple output messages) that has embedded training engines in each parallel instance. The first input stream of the operator, as described above, consists of the input events that are broadcast to all parallel instances of the operator. That is because pattern-based parallelization is chosen in the training phase with each parallel instance having a subset of the patterns. Each pattern needs to have the whole stream of events in order to search and forecast possible matches of the pattern. The reason we have chosen pattern-based parallelization is that we want to produce a single global model for every pattern.

The second input stream consists of the configuration messages. Such messages are sent by the user. Two parameters of the model to be trained can be updated on-the-fly by these messages. First, it is the order of the model, i.e., how deep into the past a state can look when calculating its transition probabilities. Second, the confidence threshold. Only forecasts with a probability higher than this threshold are emitted.

Models forwarded by the training operator are first processed by an intermidiate operator that calculates the Waiting Time Distributions of the model

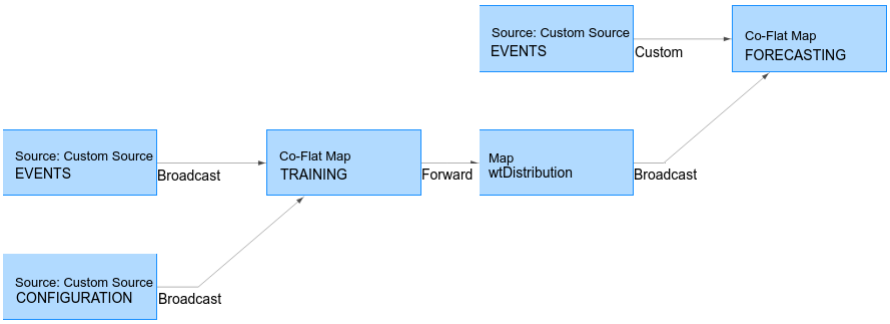


Fig. 6: Wayeb architecture.

(see Figure 6). This is a time consuming operation that increases in time the higher the new order is. As this calculation only happens whenever a new model is produced, it is detached from the rest of the training phase.

Finally, produced models reach the forecasting operator. This operator is another co-flatmap function as every input event does not guarantee a forecast. Models are broadcast to the operator because data distribution may be implemented in the forecasting phase and, in such a case, each parallel forecasting instance needs to have all the models. Moreover, each message forwarded from training has one model inside due to pattern distribution during training. Due to this, models are accompanied by their pattern id. This id is used to sort out all the models inside each parallel forecasting instance. On the other hand, input events are partitioned by a custom function, in our case a hash function on the key of the messages. This ensures that events of the same entity end up in the same parallel instance, which preserves semantics in the distributed version.

5 Experimental Evaluation

We present an extensive experimental evaluation of our parallel CEP/F engine, Wayeb, on two datasets containing *real-world* trajectories of moving objects. The first dataset comes from the domain of fleet management for vehicles moving on roads and emitting information about their status. The second dataset consists of vessel trajectories from ships sailing at sea. In both cases, our goal is to simultaneously monitor thousands of moving objects and detect/forecast interesting (or even critical) behavioral patterns in real-time, as defined by domain experts. All experiments were conducted on a server with 24 processors. Each processor is an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz. The server has 252 GB of RAM. The source code for Wayeb may be found in the following repository: `https://github.com/ElAlev/Wayeb`.

5.1 Fleet Management

Efficient fleet management is essential for transportation and logistics companies. We show how our proposed solution can effectively help in this task. With the help of experts, we define a set of patterns to be detected on real-time streams of trajectories and show that our engine can detect these patterns with an efficiency that is orders of magnitude better than real-time.

5.1.1 Dataset Description

The dataset is provided by Vodafone Innovus[4], our partner in the Track & Know project, which offers fleet management services. It contains approximately 270M records (243GB). It covers a period of 5 months, from June 30, 2018 11:00:00 PM to November 30, 2018 11:59:59 PM. The initial source emitting the data is composed of GPS (Global Positioning System) traces of moving vehicles. The data also includes speed information provided by an installed accelerometer and information regarding the level of fuel in a vehicle's tank measured by a fuel sensor. It is also enriched with weather and point-of-interest (POI) information (e.g., if a vehicle is close to a gas station, a university, a school etc), as described in [48]. Duration, accelaration and distance are some extra attributes that are calculated on the fly as they enter our system by storing information from previous events. These preprocessing steps are not necessary for using our engine. Wayeb could also work on the raw position signals. However, they do offer a wider range of behavioral patterns that can be detected, since they provide extra information which is usually not present in the raw data stream.

5.1.2 Pattern Definitions

The first pattern we have defined concerns vehicle routes. A route is the basic element of vehicle management and aggregates data between the start and the end point of a vehicle's motion cycle. A motion cycle is based on the engine status. Each vehicle route must start and end with an "engine-off" message, i.e., a message whose engine status attribute is "off". According to Vodafone Innovus, there are 12 patterns that describe the most frequent routes. These 12 route patterns can be expressed with a single Wayeb pattern as follows:

Definition 5 A route pattern for a vehicle is defined as the following sequence: emitting "engine-off" messages for at least 30 minutes, emitting at least one "moving" message and again emitting "engine-off" messages for at least 30 minutes:

$$\begin{aligned} \mathit{Route} := \; & (\mathit{Engine} = \mathit{Off} \wedge \mathit{Duration} > 30)\cdot \\ & (\mathit{Engine} = \mathit{Moving})^{+}\cdot \\ & (\mathit{Engine} = \mathit{Off} \wedge \mathit{Duration} > 30) \end{aligned}$$

[4] `https://www.vodafoneinnovus.com/`

Unfortunately, the expected data flow can be corrupted due to a variety of reasons. These reasons include bad connection during the device installation or after the vehicle has been serviced, movement of the satellites, hardware malfunctions or, simply, just an issue with the GPS. The result of these reasons is reflected in the data. For example, coordinates may change even though the vehicle is not moving or the vehicle may be moving but the coordinates remain the same. It is also often the case that the engine status is incorrect (e.g., parked messages are emitted even though engine is on, vehicle is moving yet engine status is not moving etc). These issues are important and need to be detected. We have summarized those issues in a number of patterns, defined as follows:

Definition 6 *ParkedMovingSwing*. Engine status swings between "parked" and "moving" during consecutive events.

$$\begin{aligned} ParkedMovingSwing := \ & (Engine = Parked) \cdot (Engine = Moving) \cdot \\ & (Engine = Parked) \cdot (Engine = Moving) \end{aligned}$$

Definition 7 *IdleParkedSwing*. Engine status swings between "idle" and "parked" during consecutive events.

$$\begin{aligned} IdleParkedSwing := \ & (Engine = Idle) \cdot (Engine = Parked) \cdot \\ & (Engine = Idle) \cdot (Engine = Parked) \end{aligned}$$

Definition 8 *SpeedSwing*. Speed swings between 0km/h and greater than 50km/h during consecutive events.

$$SpeedSwing := (Speed > 50) \cdot (Speed = 0) \cdot (Speed > 50) \cdot (Speed = 0)$$

Definition 9 *MovingWithZeroSpeed*. Engine status is "moving", distance traveled is greater than 30m, yet speed is 0km/h for more than 3 consecutive messages.

$$MWZS := (Engine = Moving \ \wedge \ Speed = 0 \ \wedge \ Distance > 30)^{3+}$$

Definition 10 *MovingWithBadSignal*. Vehicle is accelerating and distance traveled is greater than 30m yet there are no satellites tracking the vehicle for more than 3 consecutive messages.

$$MWBS := (Accelaration \ \wedge \ Satellites = 0 \ \wedge \ Distance > 30)^{3+}$$

In addition to the above issues, possibly related to malfunctions, experts are also interested in the following patterns:

Definition 11 *Possible Theft*. Engine status is parked, speed is 0km/h and distance traveled is greater than 30m for more than 3 consecutive messages.

$$\begin{aligned} PossibleTheft := \ & (Engine = Parked \wedge \\ & Speed = 0 \ \wedge Distance > 30)^{3+} \end{aligned}$$

Definition 12 *Dangerous Driving.* There is ice on the road and the vehicle is moving above a specific speed limit for at least 2 consecutive messages.

$$DangerousDriving := (IceExists = TRUE \wedge \ Speed > v_{limit})^{2+}$$

Definition 13 *Refuel Opportunity.* Vehicle is close to a gas station and the fuel in the tank is less than 50% for at least 2 consecutive messages.

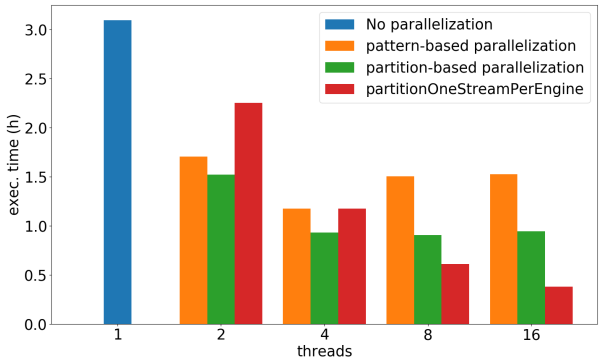
$$RefuelOpp := (CloseToGasStation = TRUE \ \wedge \ FuelLevel < 0.5)^{2+}$$

Note that Patterns 6 – 10 refer to a kind of noise detection. For our purposes, these patterns are treated as normal complex events and are simply reported whenever they are detected. We leave the decision of whether these events and their participating input events should be discarded at the discretion of analysts.

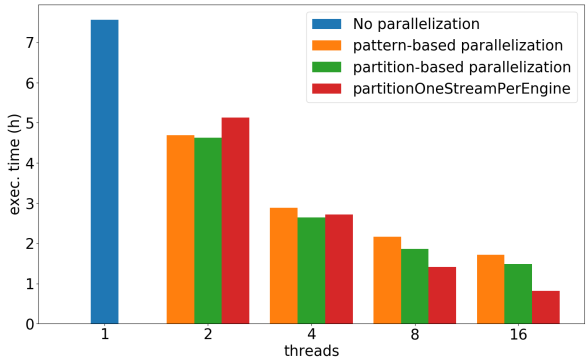
5.1.3 Recognition Results

Figure 7a and Table 2 showcase recognition times for our various parallelization techniques: pattern-based, partition-based and partition-based with one source per engine. We also show results for the non-parallel version. We have duplicated some of the patterns defined previously to simulate a greater workload of 16 patterns. We have repeated the experiment for 1, 2, 4 ,8 and 16 cores. Note that pattern-based and partition-based parallelization (i.e., orange and green bars) have an extra thread that handles the entirety of the source which is not present in the figure. This makes their actual thread count to 2+1, 4+1, 8+1, 16+1 respectively. Compared to the original single-core version, all three parallelization techniques exhibit speed-ups. For partition- and pattern-based parallelization, however, there seems to be an upper limit on the number of cores it is most efficient to use. For pattern-based parallelization there is a significant raise in time after 4 cores, while for partition-based there is no improvement after 2 cores. The reason is that the single source acts as a bottleneck. For partition-based parallelization we have one thread (the partitioner) deciding in which core each event will be forwarded, while for pattern-based events are broadcast to all cores, again in a single threaded manner. This explanation is supported by the smooth decrease in time when a parallel source is used for partition-based parallelization. While it starts off worse than pattern- and partition-based, it exhibits the best results for 16 cores.

To further support our claim above, we conducted a second experiment with a larger load, as shown in Figure 7b. 48 patterns were used this time (replicated in similar fashion as before) without any other changes. Indeed, with every recognition node having more work to do, execution time becomes less dependent on partitioning/broadcasting and more dependent on the actual recognition. Hence, speed-ups are now visible even for higher number of cores. As suspected, for parallel sources the results are not affected. Partition-based



(a) 16 patterns



(b) 48 patterns

Fig. 7: Recognition times of different parallelization techniques for different workloads. The horizontal axis represents the number of worker threads.

parallelization with parallel sources is slower when few threads are used (e.g., for two threads). This is because of the extra source thread that the other two techniques use. The work is in fact split between this one source thread and two other threads performing recognition. We thus have three threads performing similar volumes of work. When parallel sources are used, however, each parallel source is chained to a Wayeb engine in a single thread and we thus have two threads doing more work. Each thread handles half the source and performs recognition on half the stream.

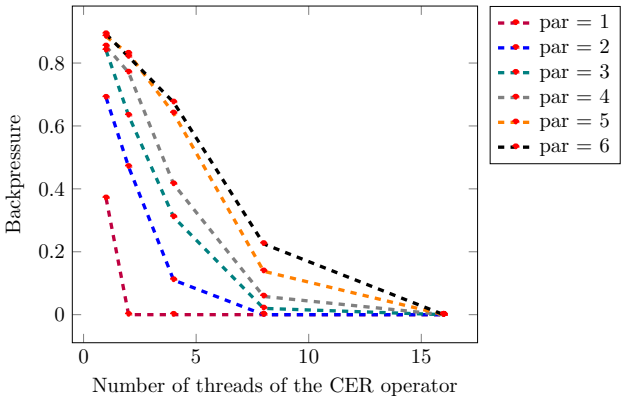
In order to evaluate recognition speed independently from source speed we had to turn operator chaining off as we cannot measure them separately when they belong in the same thread (i.e., in the case of one source per CEP engine). In addition, we leveraged parallel sources to achieve input streams of higher input rate. The goal here is to determine if our system can process input events faster than the source produces them. Flink offers a metric for this purpose, called *backpressure* [6]. Backpressure is judged on the availability of output buffers. Assuming that some task A sends events to some task B,

Table 2: Recognition times (in hours) of different parallelization techniques for different workloads.

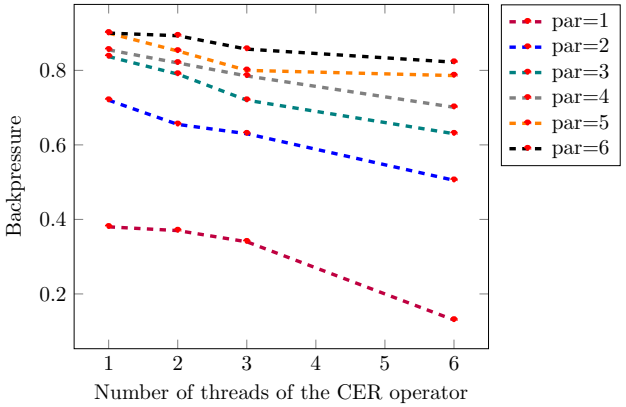
number of patterns	16	48
no parallelization	3.09	7.57
pattern-based-2-threads	1.71	4.69
partition-based-2-threads	1.52	4.63
partitionOneStreamPerEngine-2-threads	2.25	5.13
pattern-based-4-threads	1.17	2.88
partition-based-4-threads	0.93	2.65
partitionOneStreamPerEngine-4-threads	1.18	2.72
pattern-based-8-threads	1.5	2.17
partition-based-8-threads	0.91	1.86
partitionOneStreamPerEngine-8-threads	0.61	1.41
pattern-based-16-threads	1.53	1.72
partition-based-16-threads	0.94	1.49
partitionOneStreamPerEngine-16-threads	0.38	0.81

if there is no output buffer available for task A, we say that task B is back pressuring task A. In our case A, is the source operator and B is the operator with the CEP Engine. 100 samples (each sample checks if there is any output buffer available) are triggered every 50ms in order to measure backpressure. The resulting ratio notifies us how many of these samples were indicating back pressure, e.g. 0.6 indicates that 60 in 100 were stuck requesting buffers from the network stack. According to the documentation [6] a ratio between 0 and 0.1 is normal. 0.1 to 0.5 is considered to be low and anything above 0.5 is high. Note that low and high pressure will slow down the source to match the throughput of the pressuring operator.

Results for Partition-based parallelization are presented in Figure 8a. The backpressure ratio is plotted against the number of threads used for recognition. Generally, the more threads used the faster Wayeb can process events and hence less pressure is noted. Each dashed line represents a source with parallelism varying from 1 to 6. The event rate of a parallel source is measured by executing an experiment with 0% backpressure (i.e., it will not slow down due to pressure) and summing the event rate of each parallel instance presented by the flink dashboard (e.g., for 2 parallel instances of 120K events/second (e/s) for each one the overall sum is $120 + 120 = 240K$ (e/s)). Although, the greater the parallelism of the source the larger the event rate, it is not a multiplier as for 1, 2, 3, 4, 5 and 6 sources the rate becomes 130K, 240K, 350K, 430K, 440K and 490K e/s respectively. The workload in this experiment tries to emulate a real scenario and hence the route and the 5 malfunction patterns are used (i.e., they are not replicated). The results show that Wayeb can ef-



(a) Partition-based Parallelization.



(b) Pattern-based Parallelization.

Fig. 8: Backpressure experiments for fleet management. The horizontial axis expresses the number of recognition threads. Workload is 6 patterns. Each dashed line represents a different parallelism of the source stream.

fectively process streams of at least 490K e/s (black line) for this workload as 0 pressure is exhibited when 16 workers are used. As it was stated before, the duration of the dataset is 5 months which translates to roughly 13M seconds. Since the total number of input events is 270M, the average event input rate is $270M/13M \approx 20\ e/s$. Comparing the pattern throughput rate (490K e/s) with the event input rate clearly exhibits a performance that is 4 orders of magnitude better than the real-time requirements of this use case.

We perform a similar experiment for pattern-based parallelization. This time the number of threads used for recognition varies between 1, 2, 3 and 6 as there are only 6 patterns - a handicap of this technique discussed earlier. Figure 8b showcases the results of our experiment. Unfortunately, even with 6 recognition threads and a single threaded source (purple dashed line) there

is about 13% backpressure. Due to this, all sources are being slowed down to 110K e/s regardless of their parallelism. There is a drop in pressure the more recognition threads are used due to the better distribution of the patterns. However, it is not significant as the events are also multiplied as many times as the number of these threads and add extra pressure. Eventually more space is requested from the output network buffers of the source.

5.2 Maritime Situational Awareness

We now present experimental results on another real-world dataset. This dataset contains trajectories of vessels sailing at sea. We have defined a set of patterns that are similar to the ones presented in [37,39], which have been constructed with the help of domain experts. We demonstrate the effectiveness of our system which is capable of efficiently processing a dataset that contains trajectories from $\approx\ 5K$ vessels and covers a period of 6 months in less than one hour.

5.2.1 Dataset Description

A public dataset of 18M position signals from 5K vessels sailing in the Atlantic Ocean around the port of Brest, France, between October 1st 2015 and 31st March 2016 has been utilized [40]. A derivative dataset has been released in [38], containing a compressed version of the original dataset (4.5M signals), as decribed in [37]. Each trajectory in this dataset contains only the so-called critical points of the original trajectory, i.e., points that indicate a significant change in a vessel's behavior (e.g., a change in speed or heading) and from which the original trajectory can be faithfully reconstructed. We processed these compressed trajectories in order to determine the proximity of vessels to various areas and locations of interest, such as ports, fishing areas, protected NATURA areas, the coastline, etc.

5.2.2 Pattern Definitions

We now present a detailed description of the maritime patterns that we implemented, assuming that the input events contain the information described above.

Definition 14 *High Speed Near Coast*: Vessel is within 300 meters from the coast and is sailing with speed greater than 5 knots for at least one message.

$$HSNC := (IsNear(Coast) = TRUE\ \wedge\ Speed > 5)^{+}$$

Definition 15 *Anchored*: Vessel is inside an anchorage area or near a port and is sailing with speed less than 0.5 knots for at least three messages.

$$\begin{aligned} Anchored := \ & ((IsNear(Port) = TRUE \vee \\ & WithinArea(Anchorage) = TRUE) \wedge \\ & speed < 0.5)^{3+} \end{aligned}$$

Definition 16 *Drifting*: There is a difference between heading and actual course over ground greater than 30 degrees while the vessel is sailing with at least 0.5 knots for at least three messages.

$$Drifting := (|Heading - Cog| > 30 \ \wedge \ Speed > 0.5)^{3+}$$

Definition 17 *Trawling*: A vessel is inside a fishing area sailing with speed between 1 and 9 knots for at least three messages. In addition, it must be a fishing vessel.

$$\begin{aligned} Trawling := \ & (VesselType = Fishing \wedge \\ & WithinArea(Fishing) = TRUE \wedge \\ & speed > 1.0 \ \wedge speed < 9.0)^{3+} \end{aligned}$$

Definition 18 *Search and Rescue*: A SAR Vessel sails with a speed of greater than 2.7 knots and constantly changes its heading for at least three messages.

$$\begin{aligned} SAR := \ & (ChangeInHeading = TRUE \wedge \\ & VesselType = SAR \ \wedge speed > 2.7)^{3+} \end{aligned}$$

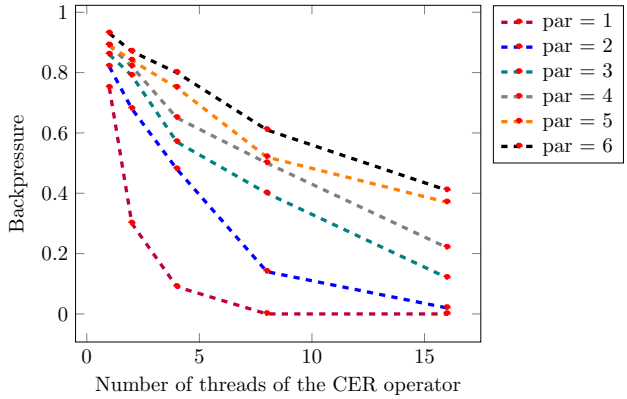
Definition 19 *Loitering*: Vessel is neither near port nor the coastline while it sails with speed below 0.5 knots.

$$\begin{aligned} Loitering := \ & (IsNear(Port) = FALSE \wedge \\ & IsNear(Coast) = FALSE \wedge \\ & speed < 0.5)^{3+} \end{aligned}$$

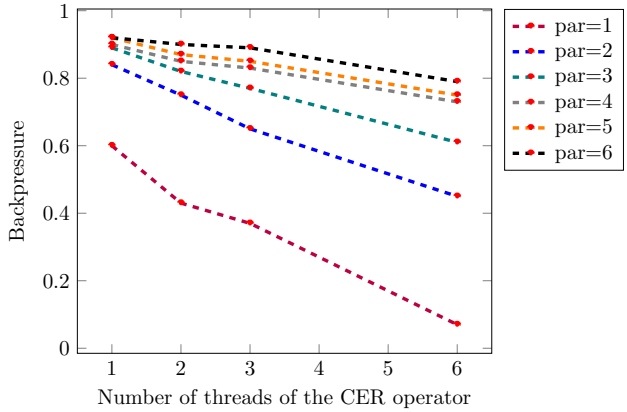
5.2.3 Recognition Results

We used the 6 patterns defined above as the workload for a number of experiments. Following a similar approach to our fleet management experiments, we avoided replicating the patterns to emulate a real scenario and evaluated them for streams of different event rates.

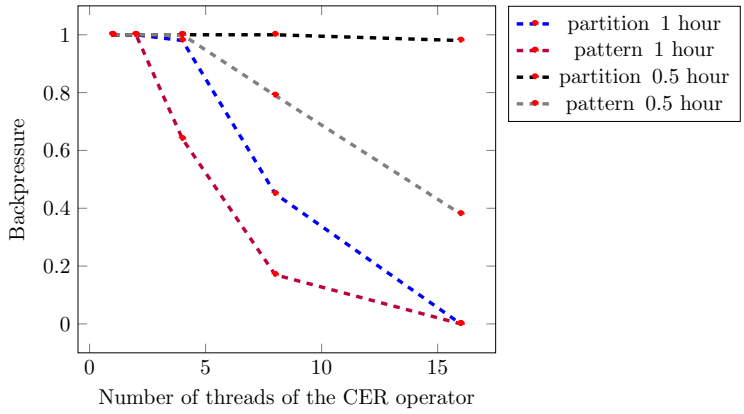
Figure 9a presents results for the partition-based parallelization scheme. This time the backpressure remains always above 40% for a 6-threaded input source, even with 16 recognition threads. Due to the fact that the CEP operator cannot keep up with the initial rate of the source, the source has to adjust its rate to match the throughput of the CER operator. According to the Flink dashboard, this rate is at most $180K$ e/s (with 16 worker threads). This lower throughput of our CEP operator (compared with the fleet management use case) can be attributed to the fact that the patterns are now more complex, as



(a) Partition-based backpressure for a workload of 6 patterns. Each dashed line represents a different parallelism of the source stream. Event rate is capped at 180K e/s due to backpressure.



(b) Pattern-based backpressure for a workload of 6 patterns. Each dashed line represents a different parallelism of the source stream. Event rate is capped at 90K e/s due to backpressure.



(c) Partition- and pattern-based parallelism for the dataset being replayed in one 1 and 0.5 hours respectively. Workload is 220 patterns of vessels approaching 220 different ports.

Fig. 9: Backpressure experiments for the Maritime dataset. The horizontal axis expresses the number of recognition threads.

on average they contain more unions, disjunctions and iterations to evaluate for every event. The duration of the dataset is 6 months which translates to roughly 15.5M seconds. Since the total number of input events is about 4.5M, the average event input rate is $15.5M/4.5M \approx 3.5\ e/s$. Comparing the pattern throughput rate with the event input rate showcases again a performance that is 4 orders of magnitude better than the real-time requirements of this use case. The results for pattern-based parallelism are presented in Figure 9b and follow a similar trend. The event rate here is capped at $90K\ e/s$ due to backpressure.

We conducted a second series of experiments with a setting where the number of patterns is naturally high, in order to determine whether pattern-based parallelism offers an advantage in such settings. Consider the following pattern, describing the movement of a vessel as it a approaches a port.

Definition 20 *Approaching Port*: Vessel is initially more than 7 km away from the port, then, for at least one message, its distance from the port is between 5 and 7 km and finally it enters the port (i.e., its distance from the port falls below 5 km).

$$\begin{aligned} Port := \ & (DistanceToPort(Port_X) > 7.0) \wedge \\ & DistanceToPort(Port_X) < 10.0) \cdot \\ & (DistanceToPort(Port_X) > 5.0) \wedge \\ & DistanceToPort(Port_X) < 7.0)^{+} \cdot \\ & (DistanceToPort(Port_X) < 5.0) \end{aligned}$$

The predicate *DistanceToPort* calculates the distance of a vessel from the port $Port_X$ passed as argument and is evaluated online. If we want to monitor vessel activity around every port in a given area, then we need to replicate this pattern N times, if there are N distinct ports. We would thus naturally have N patterns, which would be almost identical except for the argument passed to *DistanceToPort* ($Port_1$, $Port_2$, up to $Port_N$). For the area of Brest, the total number of ports is 220. We run an experiment with these 220 patterns with partition- and then with pattern-based parallelization. Figure 9c shows the results. Contrary to previous experiments, in this one we used a stream simulator to feed the dataset to our CEP system. This simulator, instead of reading input events from a file and instantly sending them to our engine, has the ability to insert a delay between consecutive events. For example, we can set the delay to be exactly the time difference between two events. This would allow us to re-play the stream as it was actually produced, which would take 6 months for this dataset. We also have the ability to re-play the stream at higher speeds. For these experiments, we re-played the stream at various different speeds in order to determine the "breaking point" of our system. Figure 9c shows the results for two such speeds, where the whole stream was processed in 0.5 and 1 hour, corresponding to a speed-up of $x8640$ and $x4320$ compared to the original dataset. While the CEP operator lags behind the source when it is re-played at half an hour, it is evident that it can process it

without any problems when it is replayed at one hour, as both pattern- and partition-based parallelism exhibit 0% backpressure. In fact, pattern-based parallelism performs better in this experiment. This lends credence to our belief that pattern-based parallelism might actually be more suitable than partition-based parallelism when there is a high number of patterns to be processed simultaneously.

5.3 Forecasting Experiments

For the forecasting experiments described in this Section, we focus on the *CLASSIFICATION-NEXTW* task (see Section 3.2). The evaluation task itself consists of the following steps. At the arrival of every new input event, we first move to a new automaton state, as explained above. We use its waiting-time distribution to produce the forecast. Two parameters are taken into account: the length of the future window w within which we want to know whether a CE will occur and the confidence threshold θ_{fc}. If the probability of the first w points of the distribution exceeds the threshold θ_{fc}, we emit a positive forecast, essentially affirming that a CE will occur within the next w events; otherwise, we emit a negative forecast, essentially rejecting the hypothesis that a CE will occur. We thus obtain a binary classification task.

As a consequence, we can make use of standard classification measures, like precision and recall. Each forecast is evaluated: a) as a *true positive* (TP) if the forecast is positive and the CE does indeed occur within the next w events from the forecast; b) as a *false positive* (FP) if the forecast is positive and the CE does not occur; c) as a *true negative* (TN) if the forecast is negative and the CE does not occur and d) as a *false negative* (FN) if the forecast is negative and the CE does occur; Precision is then defined as $Precision = \frac{TP}{TP+FP}$ and recall (also called sensitivity or true positive rate) as $Recall = \frac{TP}{TP+FN}$.

There are also some other metrics that can be used for assessing the quality of the forecasts, such as the Root Mean Squared Error (RMSE) and the Mean Absolute Error (MAE) (MAE is less sensitive than RMSE to outliers). The so-called *negatively oriented interval score* (NOIS) [28] is another such metric for evaluating forecasting given in the form of intervals (RMSE and MAE assume that forecasts refer to a single timepoint). NOIS penalizes forecasts whose intervals are long (so that focused intervals are promoted). If a forecast is correct, then no other penalty is applied. If it is false, then an extra penalty is added, which is essentially the deviation of the forecast from the actual observation, weighted by a factor that grows with the confidence of the forecast. We avoid using these metrics in this paper, because they can only be applied in regression experiments, since they all assume that there is an observation (i.e, a complex event occurs) and the timestamp of this observation can be compared to our forecast. As we have explained, we currently refrain from running regression experiments.

For the following forecasting experiments we have used the approaching port pattern for the port of Brest. The results of partition-based parallelization

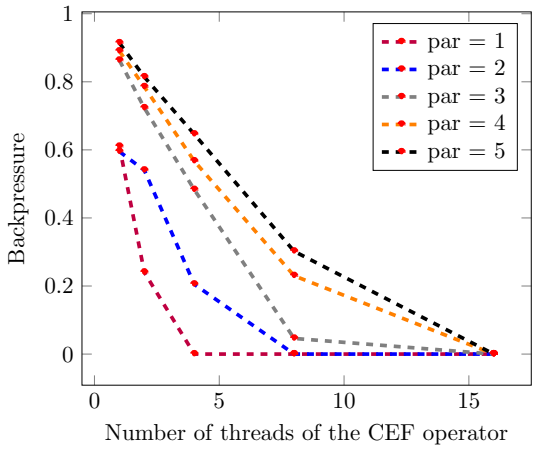
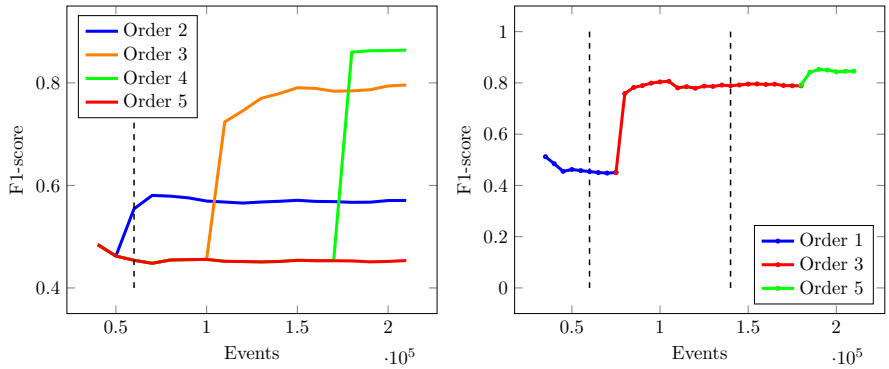


Fig. 10: Maritime backpressure experiment of vessels approaching the port of Brest. Partition-based parallelization is used. The horizontal axis expresses the number of forecasting threads. Each dashed line represents a different parallelism of the source stream

for forecasting are presented in Figure 10. The backpressure ratio is plotted against the number of threads used for forecasting. Each dashed line represents a source with parallelism varying from 1 to 5. Event rate of 1, 2, 3, 4 and 5 sources is calculated at 100K, 200K, 285K, 348K and 375K e/s respectively. Again, note that the parallelism of the source is not a multiplier of the event rate, The workload in this experiment tries to foresee which vessels approach the port of Brest. The results show that Wayeb can effectively process streams of at least 375 e/s (black line) for this workload as 0 pressure is exhibited when 16 workers are used. As with the recognition experiments the event rate of the stream is $\approx 3.5\ e/s$. Comparing the pattern throughput rate (375K e/s) with the event input rate clearly exhibits a performance that is 5 orders of magnitude better than the real-time requirements of this use case.

We now demonstrate how the adaptation mechanism works when we update the order of a model. Changing the order via a configuration message requires the construction of a new Counter Suffix Tree (CST) (see Section 3.2). This new CST however does not have any information about the previous events of the input stream. Thus, we need to wait for a specific number of events to be processed in the training phase before actually producing the new model. This number of events is user-defined and in this experiment it is 30K events. After the required number of events is met the calculation of the Waiting Time Distributions (WTD) takes place. The higher the order the more time it takes for the calculation to complete. Figure 11 showcases this behavior. Figure 11a shows results where configuration messages are sent at 30K events. The calculation of WTD happens at 60K events (dashed vertical line) due to the extra heating period of 30K events needed for training. The



(a) Messages sent at 30K messages.

(b) Changing the order on-the-fly.

Fig. 11: On-the-fly order adaptation. Dashed lines indicate the points in the stream when the message to change the order is sent.

change in F1 score values implies that the new model has been created and forwarded to the forecasting operator. Originally the order is 1 and the experiment is performed multiple times, each time with a different order. For an order of 2 the model is instantly updated at 60K events. For order 3 it takes about 50K events to calculate the WTD. For 4 it takes 110K events. Finally, for an order of 5 execution time is so large that the whole dataset is processed by the forecasting operator before updating the model. Thus, we don't see any change in the F1-score.

Figure 11b shows results from experiments where an attempt is made to change the order of the model twice at runtime. A stream simulator was used which emits the whole dataset in about a minute, rather than consume it instantly from Kafka. Calculations of WTD start at 60K messages and at 140k messages for orders of 3 and 5 respectively. As in the previous experiments, we witness an increase in the F1-score when the order increases.

For completeness, we show how the throughput is affected as the order of the model changes. Figure 12 shows that it largely remains unaffected. Originally, it starts from about 7K e/s and rises from there on. This has to do we the JVM heating up and compiling the code. It has nothing to do with the order. After this raise it hovers at about 12K e/s. To simplify the presentation, we have used a non-parallel version, hence the throughput values are rather low.

Finally, we have run experiments to showcase the adaptation of the threshold parameter. Figure 13 showcases the results. After the first 30K events have been processed by the training operator a model is produced with some threshold value. Messages to change this value are sent at both the 60,000th event and 140,000th event. There is a forced pause on the stream during these two events in order to wait for the waiting-time distributions to be calculated so models are seemingly updated instantly. We do this to simplify the presentation as WTDs execution times are unaffected when the order remains the

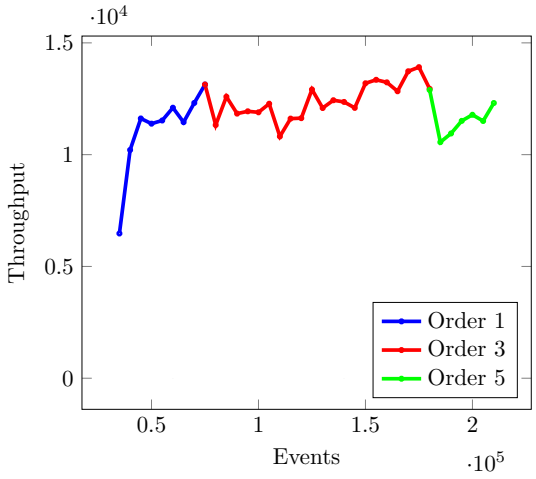
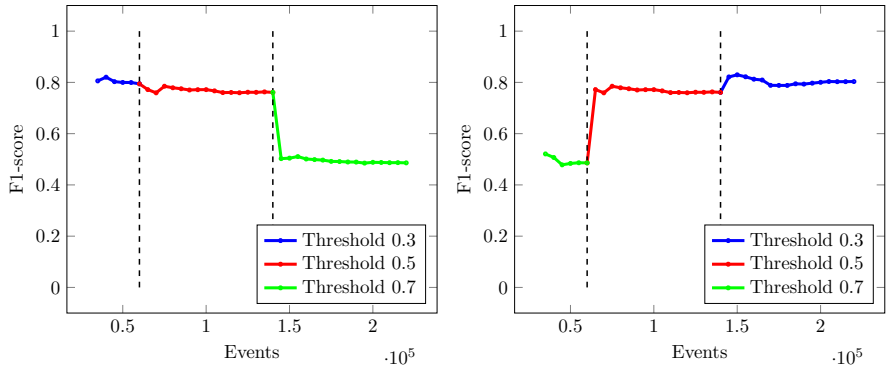


Fig. 12: Throughput in the case of on-the-fly order adaptation.



(a) Changing the threshold in an ascending fashion.

(b) Changing the threshold in a descending fashion.

Fig. 13: On-the-fly threshold adaptation

same. Moreover, when updating the threshold of a model, there is no need to create a CST from scratch (like we did with the order parameter) and hence wait for a fixed number of events to be processed by the new model. Instead, the same CST is used which means the later we update the threshold of a model the more events this model will have under its belt.

Figure 13 measures how this number of events a model has processed affects performance. By using an ascending and a descending fashion to the updating threshold values in Figures 13a and 13b respectively, we test if models with the same threshold value but having processed different number of events exhibit different F1-scores. In this case it seems that using 30K events only for training is sufficient. When using a threshold of 0.3 (blue line in both figures) both the

model that has processed 30K events (Figure 13a) and the other model with 140K events (Figure 13b) exhibit similar F1 scores (around 80%). The same holds for the 0.7 threshold value (green line in both figures) as both the 30K and 140K versions of the model have around 40% F1-score.

6 Summary & Future Work

We presented Wayeb, an open-source tool for Complex Event Recognition and Forecasting, as a means of analyzing big mobility data streams. We defined a number of patterns that are useful in fleet management and maritime monitoring applications. Moreover, we presented implementations of two parallelization techniques and compared their efficiency against the single-core version. Our results demonstrate the superiority of partition-based over pattern-based parallelization, when the number of patterns is relatively low. When this number is high, then pattern-based parallelization becomes a viable option.

Furthermore, we introduced a parallel version for online training and forecasting enhanced with adaptation capabilities. Forecasting experiments showcased results similar to recognition, as the throughput of partition parallelization is orders of magnitude higher than the event rate of the input stream. Our preliminary adaptation experiments showed that increasing the order of a model leaves forecasting throughput unaffected, while it greatly increases accuracy. However this comes at a cost of training execution time as there is a delay in producing the new model.

For the future, we intend to combine various parallelization techniques and construct more patterns for the domains presented. Another research avenue would be to compare our automata-based method against other approaches, such as logic-based ones, which have been applied to similar datasets [37, 48]. Finally, we aim to improve the adaptation process and have the order, threshold (and possibly other) parameters be set automatically rather than having the user send them. For example, Bayesian optimization could possibly be used in order to find the sweet spots for the values of these parameters that try to maximize the accuracy of our forecasts and the throughput of our system while also trying to minimize training time.

Acknowledgements This work was funded by European Union's Horizon 2020 research and innovation programme Track & Know "Big Data for Mobility Tracking Knowledge Extraction in Urban Areas", under grant agreement No 780754. It is also supported by the European Commission under the INFORE project (H2020-ICT- 825070).

References

1. Apache flink - stateful computations over data streams. `https://flink.apache.org/`
2. Apache kafka. `https://kafka.apache.org/`
3. Esper. `http://www.espertech.com/esper`
4. Esperonstorm. `https://github.com/tomdz/storm-esper`

5. Flinkcep - complex event processing for flink. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html
6. Monitoring back pressure. https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/back_pressure.html
7. Siddhi cep. https://github.com/wso2/siddhi
8. Task chaining and resource groups. https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/\#task-chaining-and-resource-groups
9. Wso2. creating a storm based distributed execu-tionplan. https://docs.wso2.com/display/CEP410/Creating+a+Storm+Based+Distributed+Execution+Plan
10. Abe, N., Warmuth, M.K.: On the computational complexity of approximating distributions by probabilistic automata. Machine Learning **9**, 205–260 (1992)
11. Alevizos, E., Artikis, A., Paliouras, G.: Event forecasting with pattern markov chains. In: DEBS (2017)
12. Alevizos, E., Artikis, A., Paliouras, G.: Wayeb: a tool for complex event forecasting. In: LPAR (2018)
13. Alevizos, E., Artikis, A., Paliouras, G.: Complex event forecasting with prediction suffix trees. VLDB J. (2021)
14. Alevizos, E., Skarlatidis, A., Artikis, A., Paliouras, G.: Probabilistic complex event recognition: A survey. ACM Comput. Surv. **50**(5), 71:1–71:31 (2017)
15. Artikis, A., Sergot, M., Paliouras, G.: An event calculus for event recognition. IEEE Trans. Knowl. Data Eng. (2015)
16. Balkesen, C., Dindar, N., Wetter, M., Tatbul, N.: RIP: run-based intra-query parallelism for scalable complex event processing. In: DEBS (2013)
17. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order markov models. J. Artif. Intell. Res. **22**, 385–421 (2004)
18. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink^{TM}: Stream and batch processing in a single engine. IEEE Data Eng. Bull. (2015)
19. Christ, M., Krumeich, J., Kempa-Liehr, A.W.: Integrating predictive analytics into complex event processing by using conditional density estimations. In: EDOC Workshops, pp. 1–8. IEEE Computer Society (2016)
20. Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. IEEE Trans. Communications **32**(4), 396–402 (1984)
21. Cugola, G., Margara, A.: Complex event processing with T-REX. J. Syst. Softw. (2012)
22. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. (2012)
23. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: CAV (1) (2017)
24. Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.: Cayuga: A general purpose event monitoring system. In: CIDR (2007)
25. Engel, Y., Etzion, O.: Towards proactive event-driven computing. In: DEBS, pp. 125–136. ACM (2011)
26. Fülöp, L.J., Beszédes, Á., Toth, G., Demeter, H., Vidács, L., Farkas, L.: Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics. In: BCI, pp. 26–31. ACM (2012)
27. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.: Complex event recognition in the big data era: a survey. VLDB J. (2020)
28. Gneiting, T., Raftery, A.E.: Strictly proper scoring rules, prediction, and estimation. Journal of the American Statistical Association **102**(477), 359–378 (2007)
29. Hirzel, M.: Partition and compose: parallel complex event processing. In: DEBS (2012)
30. Koutroumanis, N., Santipantakis, G., Glenis, A., Doulkeridis, C., Vouros, G.: Integration of mobility data with weather information. In: EDBT/ICDT Workshops (2019)
31. Li, Y., Ge, T., Chen, C.X.: Data stream event prediction based on timing knowledge and state transitions. Proc. VLDB Endow. **13**(10), 1779–1792 (2020)
32. Liu, M., Rundensteiner, E., Greenfield, K., Gupta, C., Wang, S., Ari, I., Mehta, A.: E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In: SIGMOD (2011)

33. Mei, Y., Madden, S.: Zstream: a cost-based query processor for adaptively detecting composite events. In: SIGMOD (2009)
34. Muthusamy, V., Liu, H., Jacobsen, H.: Predictive publish/subscribe matching. In: DEBS, pp. 14–25. ACM (2010)
35. Ntoulias, E., Alevizos, E., Artikis, A., Koumparos, A.: Online trajectory analysis with scalable event recognition. In: EDBT/ICDT Workshops, *CEUR Workshop Proceedings*, vol. 2841. CEUR-WS.org (2021)
36. Pandey, S., Nepal, S., Chen, S.: A test-bed for the evaluation of business process prediction techniques. In: CollaborateCom, pp. 382–391. ICST / IEEE (2011)
37. Patroumpas, K., Alevizos, E., Artikis, A., Vodas, M., Pelekis, N., Theodoridis, Y.: Online event recognition from moving vessel trajectories. GeoInformatica (2017)
38. Patroumpas, K., Spirelis, D., Chondrodima, E., Georgiou, H., P, P., P, T., S, S., N, P., Y, T.: Final dataset of Trajectory Synopses over AIS kinematic messages in Brest area (ver. 0.8) [Data set] (2018). URL `http://doi.org/10.5281/zenodo.2563256`
39. Pitsikalis, M., Artikis, A., Dreo, R., Ray, C., Camossi, E., Jousselme, A.: Composite event recognition for maritime monitoring. In: DEBS (2019)
40. Ray, C., Dreo, R., Camossi, E., Jousselme, A.: Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance (2018). URL `https://doi.org/10.5281/zenodo.1167595`
41. Ron, D., Singer, Y., Tishby, N.: The power of amnesia. In: NIPS, pp. 176–183. Morgan Kaufmann (1993)
42. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. Machine Learning **25**(2-3), 117–149 (1996)
43. Sakr, M.A., Güting, R.H.: Spatiotemporal pattern queries. GeoInformatica **15**(3), 497–540 (2011)
44. Sakr, M.A., Güting, R.H.: Group spatiotemporal pattern queries. GeoInformatica **18**(4), 699–746 (2014)
45. Schultz-Møller, N., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: DEBS (2009)
46. Snidaro, L., Visentini, I., Bryan, K.: Fusing uncertain knowledge and evidence for maritime situational awareness via markov logic networks. Inf. Fusion (2015)
47. Terroso-Saenz, F., Valdés-Vela, M., Skarmeta-Gómez, A.: A complex event processing approach to detect abnormal behaviours in the marine environment. Inf. Syst. Frontiers (2016)
48. Tsilionis, E., Koutroumanis, N., Nikitopoulos, P., Doulkeridis, C., Artikis, A.: Online event recognition from moving vehicles: Application paper. TPLP (2019)
49. Willems, F.M.J., Shtarkov, Y.M., Tjalkens, T.J.: The context-tree weighting method: basic properties. IEEE Trans. Information Theory **41**(3), 653–664 (1995)
50. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: SIGMOD (2014)