INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Concurrent Stream Reasoning for Complex Event Recognition

Florian Eich

Supervisor: Prof. Dr. François Bry
Mentor: Dipl.-Ing. Thomas Prokosch
Submitted: 2022-09-15

Florian Eich:
Concurrent Stream Reasoning for Complex Event Recognition

September 15, 2022

SUPERVISOR:
Prof. Dr. François Bry (LMU)

MENTOR:
Dipl.-Ing. Thomas Prokosch

LOCATION:
Munich, Germany

Abstract

Complex Event Recognition (CER) is concerned with detecting patterns of interest in data streams which are collections of data points with temporal order. These data streams have become ubiquitous in recent years with the advancement of global digitalization: e-Commerce and social media platforms have seen growing user bases; electronic sensors have become more affordable; vehicles and all manner of appliances are now internet connected data sources.
At the heart of CER is the concept of an *event* which itself is based on the notion of a *state*. A *state* is a configuration of statements which can be either true or false that exists at a distinct point in time. An event is something which causes a state to change, i.e. something which causes a *transition*. The Event Calculus (EC) as presented by Kowalski and Sergot (1986) provides a suitable logic-based formalization and makes it possible to work with the aforementioned concepts. The declarative nature of the EC enables powerful and ergonomic expressions which is a distinctive advantage when it comes to the integration of domain expert knowledge. Implementations and dialects of the EC such as the Event Calculus for Run-Time Reasoning (RTEC) by Artikis et al. (2015) have aimed to extend this advantage while making the formalization viable for uses cases requiring real-time capabilities.
Although these efforts have not been without success, they have also found limitations: to achieve their goal, they have had to make use of extensive preprocessing, cacheing of results and limiting the scope of the analysis. This thesis presents a concurrency focused formalization of the EC anad a concurrently executable implementation of a rule-based processing architecture modeled on it, thus providing a way forward to overcome these limitations more elegantly.
The approach allows for better resource usage on modern systems while retaining the expressive power of a logic-based stream reasoning system. A publisher-subscriber architecture in which nodes implement event recognition functions is used to implement a prototype for maritime vessel monitoring based on previous works. This implementation outperforms sequential execution by an order of magnitude while retaining good ergonomics for integrating domain expert knowledge.

Declaration of Originality

I hereby declare that I have produced this work without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

Munich, September 12, 2022

. . . . . .

Florian Eich

Acknowledgements

I would like to express my gratitude to Mr. Thomas Prokosch, who, showing patience and empathy I admire deeply, guided me through the struggle of producing this thesis. I want also to thank Prof. Dr. François Bry, who accepted this thesis knowing of my situation and with a steady hand helped me reach the finish line.

Furthermore, I am grateful to my work colleagues Helmut Mravlag and Cinzia Piacquadio, who took over a great deal of my responsibilities in order to free up my time to write this thesis.

Lastly, I thank Lucrezia Carnelos for her support and understanding, and her ability to rekindle my confidence when it was fading.

Contents

1 Introduction

> *My hypothesis is that we can solve [the software crisis in parallel computing], but only if we work from the algorithm down to the hardware — not the traditional hardware first mentality.*
>
> — Tim Mattson, Principal Engineer at Intel[1]

Although a precise definition is difficult to come by, *data* can be thought of as a collection of symbols with a relation to the physical world whereas *information* is considered to be data which has been processed with the goal of drawing conclusions and enabling decision making. Techniques for processing data — *turning data into information* — and their characteristics are therefore the subject of intensive research and development. This thesis focuses on the processing of *data streams* using logic-based evaluation in a concurrent manner to recognize *complex events* efficiently.

A data stream is a temporally ordered tuple of data points that is extended with new data points as time passes. As such, data streams are present in any system in which the time domain is a core characteristic. Examples are newspaper articles and media feeds in general, social media platforms, sensory measurements in devices from servers to vehicles, and many more. The number of data streams sources is growing, and the amount of data with it. In fact, the ubiquity and importance of data streams has all but exploded in recent decades and the challenges that arise from them have followed suit.

Consider the following primitive example. If a vehicle records ten 32-bit floating point values every second, it records a total of roughly 1.2GB in a year. This may not sound significant at first glance, but taking into account the many millions of vehicles operating in even geographically very limited areas such as cities, it is clear that data streams reach orders of magnitude in size where efficient processing is required to turn data into information.

One kind of information that corresponds to data streams are *events*. The Oxford English Dictionary defines an event as *A thing that happens or takes place, especially one of importance*. The nature of an event is temporal — *A thing that happens* happens at a certain point in time. The information about an event has therefore two components: what it is and when it happens (or happened), e.g. *Julius Caesar died* (what) *on March 15, 44 BC* (when). The processing task of finding events in data streams is called Complex Event Recognition (CER). There is no universally accepted metric for what constitutes a *complex* event, but a commonplace definition is that simple events are detected from data directly whereas complex events are composed of any number of simple events.

In their 2020 paper *Complex Event Recognition in the Big Data Era: A Survey* [1], Giatrakos et al. categorize several different approaches used in current CER systems: automata-based systems, logic-based systems, tree-based systems and hybrid approaches. The authors find that logic-based systems exhibit good overall characteristics; in particular, logic-based systems enable an intuitive definition of rules to recognize events and do not impose theoretical limitations on concurrency (as for example automata-based systems do, according to the authors).

Logic-based approaches for CER are an active research topic and show good results. However, they do have known limitations. Proving their efficacy often requires labelled datasets and while domain expert knowledge is easier to integrate, doing so often requires a fair share of expertise from the domain experts in terms of using the integration. Existing implementations also sometimes struggle to achieve good computational efficiency on modern systems.

In this thesis, a logic-based approach is considered in detail, most notably the approach implemented by Kowalski and Sergot in their Event Calculus (EC) [2]. The EC provides a formalization which allows for the specification of rules based on domain expert knowledge to recognize events on data streams. Since the presentation of the EC in 1986, a number of extensions and dialects to it have been developed, most notable the RTEC which has been presented by Artikis et al. in their 2015 paper *An Event Calculus for*

[1] Peter Clarke, EETimes: *Ten quotes on parallel programming*, https://www.eetimes.com/ten-quotes-on-parallel-programming/

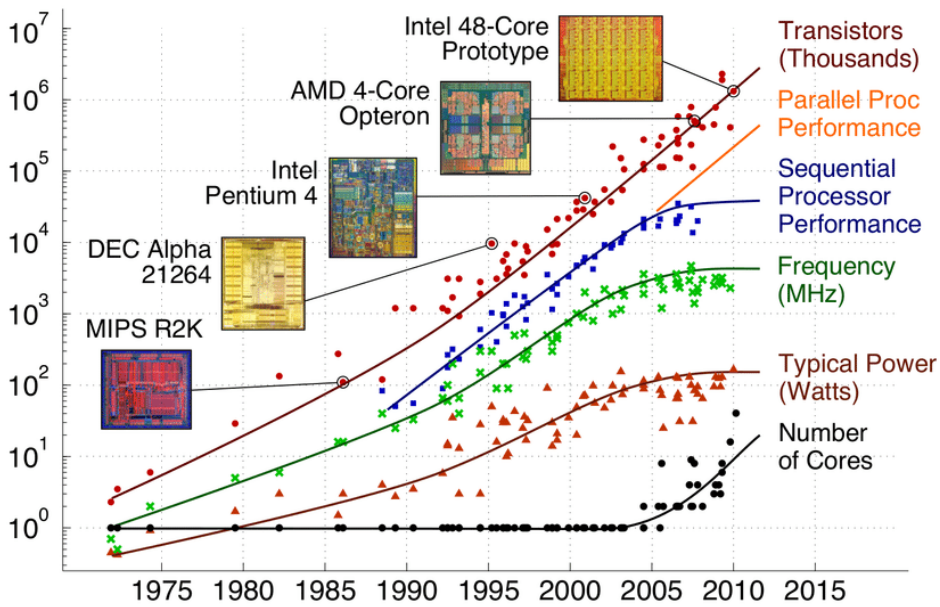

Figure 1.1.1: Evolution of multi-core processors [7]

Event Recognition [3]. The authors of this and other works based on RTEC [4, 5] show its viability for real-time event recognition.
Revisiting the above example of a single vehicle recording sizeable amounts of data every year and what this means for the amounts of data produced by fleets of many vehicles, it stands to reason that making efficient use of modern computing systems is required for an approach to be viable today. Previous works have shown the general viability of the logic based approach but have shown its limitations when executing the evaluation sequentially. Artikis et al. [3] mention the use of cacheing and limiting the scope of evaluation ('windowing') to achieve real-time capability. Patroumpas et al. [6], in an effort where a similar dataset was used, make heavy use of data preprocessing before evaluating using RTEC, which itself uses windowing, for the same reason. Pitsikalis et al. [4] utilize similar methods and shortly discuss the potential opportunities of trivial parallelization with the goal of increasing the efficiency of their implementation. This thesis sets out to realize a concurrent approach based on the EC to do exactly that whilst retaining the ability for domain expert knowledge to be integrated with a reasonable effort.

1.1 Motivation

CER on data streams is often time critical, as events must be recognized in real time (i.e. before a given deadline) in order to be useful. If, for example, the CER system detects potentially illegal behavior and it is the goal to use its results to apprehend perpetrators, the results must be available in time for law enforcement operators to react. Striving towards the efficient use of modern computing resources is therefore more than a mere academic endeavour.
In his 2014 thesis, Tendulkar evaluates the evolution of multi-core processors [7]. As Figure 1.1.1 (taken from said thesis) shows, the number of cores per processor has been increasing since the 2000s, thus making parallel execution of computational tasks a necessity for efficient use of modern systems. Concurrency as the most generic approach to making a single workload execute on cores working in parallel is therefore a popular choice for modern architectures and is widely used.
Furthermore, CER presents opportunities particularly in fields where domain expert knowledge is abundant. Any solution must allow for its integration with reasonable effort in order to be viable. Previous implementations based on RTEC have achieved this elegantly as they make it possible for rules which have been developed by domain experts to be declared in Prolog [3], a programming language designed specifically for logic programming [8].

This thesis explores the possibilities and opportunities of executing logic based event recognition concurrently. Furthermore, this thesis presents an ergonomic interface to implement concurrent logic-based CER on data streams written in Rust. In order to assess the potential of the approach and test the implementation, we make use of prior work presented by Artikis et al. in *An Event Calculus for Event Recognition* [3] and by Pitsikalis et al. in *Composite Event Recognition for Maritime Monitoring* [4]. The authors present (a) an implementation of the event calculus, (b) the implementation of a CER system using their implementation of the event calculus and (c) a formalized collection of the domain expert knowledge required to test the approach using a publicly available dataset of maritime vessel locations and trajectories [9]. Using their material, we measure the performance of our approach and thus determine its viability.

1.2 Data streams, stream reasoning and concurrency

Data streams In Chapter 1, a data stream is defined as a tuple of temporally ordered data points which is extended with new data points over time. A data point in this context is an object with the following components:

- an identifier denoting the data stream it belongs to
- an identifier denoting the time of its recording, typically called *timestamp*
- a value or a collection of values

The value or collection of values is the component which varies over time. As such, it is obvious that any data which has a temporal component can be represented as a data stream, whether it is the movement of a tennis ball as it bounces on the ground or a sequence of news articles which is published on a subject over time.
A common source of data streams are sensory measurements. Weather stations are perhaps the most straightforward example, but also industrial facilities and machinery, information technology infrastructure and many kinds of vehicles are equipped with sensors which are used to acquire measurements at given time intervals, thus producing data streams. In the context of this work, streams of data points which contain location, speed and status values collected by maritime vessels are considered.

Stream reasoning Barbieri et al. define stream reasoning as "logical reasoning in real time on [...] data streams [...]" ([10], p. 2). The term *logical reasoning* refers to the use of logic functions, which are functions that apply rules which evaluate to a boolean value to data points. Such functions are typically represented using a symbolic formalization, e.g. Horn Clauses. The term *real time* refers to results of data processing being available within a given deadline, which is either imposed by the frequency with which the data stream is updated with new data points or by an external requirement.
As such, stream reasoning is a form of data processing which incorporates the knowledge of domain experts in a formalized way. Logic functions which model the knowledge are defined and applied to data streams, resulting in the detection of conditions which are relevant to the task which the process is designed to support. In the context of this work, a distinction is made between functions which have a numeric or other application specific value as their output and functions which have a boolean value as their output; both are required for CER.

Concurrency Lamport defines the relation "$a \rightarrow b$" between two events a and b (the definition for *event* as *A thing that happens (at a certain point in time)*, provided in Chapter 1, shall suffice) to mean that a comes before b (if both events are handled by the same computational process) or that a is the "send" event for a message for which b is the "receive" event (if the events are handled by separate computational processes) [11]. This definition may be rephrased as "$a \rightarrow b$" meaning that a must happen before b, or that b has a temporal dependency on a. Subsequently, a and b are said to be *concurrent* if no temporal dependency exists between them, or "$a \nrightarrow b$".
It follows that in order to handle two events a and b in parallel, they must be concurrent. If they are not, they must be handled in sequence. In the context of this work, the concurrent application of stream reasoning to data streams with the goal of enabling parallel execution is considered.

1.3 Notation and terminology

Although there are commonalities, there is no canonical terminology for the components of a CER system. This is due to a number of factors, most notably the multitude of different approaches which in turn lead to distinct requirements when it comes to the terminology that is used to describe them. This work is guided by the original work on the Event Calculus [2], an article on the Event Calculus written by Shanahan [12], the canon of work done on RTEC by Artikis et al. [3] and subsequent papers and lastly the *Event Processing Glossary* by Clark et al. [13]. The following terms and definitions are adapted and partially refined from those sources and are used throughout the monograph.
There are two concepts related to time which must first be specified:

1. time instant: discrete point in time, given as an approximate time information or simple index
2. time interval: span in time, usually represented by a 2-tuple consisting of two time stamps which define the beginning and the end of the interval respectively

Other widely used terms and concepts require a definition for this work as they are use ambiguously elsewhere. The following terms are defined based on the concepts of *time instant* and *time interval* specified above and hierarchically based on each other:

- A *state*, according to the Oxford English Dictionary, is *the particular condition that somebody or something is in at a specific time*. In the context of this work, the state is the collection of data and information items known at any given time instant.
- A *fluent* is any property that varies over time. This can be a value in a data stream as described in Section 1.2 or the output of a function which is evaluated to obtain the value of a fluent.
- A *Complex Event (CE)* is a fluent which can only have the values `true` and `false`. In the context of this work, CEs are expressed as predicate clauses of fluents.

Lastly, CER systems can rely on information which is not gathered from data stream but rather known beforehand, such as geographical information or generic meta information that relates to the domain in which the CER system is applied. Such information is referred to under the term *background knowledge* in this work. According to Giatrakos et al. [1], the integration of background knowledge is a capability that is not shared by all CER systems, however the implementation called RTEC and presented by Artikis et al. [3] as well as all implementations based on it have this capability.

1.4 Structure and scope of this thesis

In this chapter, an overview of the treated subject as well as insight into some previous work and concepts has been provided. In Chapter 2, a more detailed overview of related work is provided alongside an analysis of the limitations found there. Chapter 3 presents the formalization of the EC for enabling concurrency proposed in this thesis, with Chapter 4 providing a detailed look into the prototypical implementation of the formalization. An experimental evaluation of the implementation is presented in Chapter 5. This thesis concludes with a review, focusing on core aspects and future opportunities in Chapter 6.

2 Related work

Several formalizations have been presented in the literature for reasoning over data streams with the purpose of Complex Event Recognition (CER). One of them called *Event Calculus (EC)* is the subject of an article published by Robert Kowalski and Marek Sergot in 1986, titled *A Logic-based Calculus of Events* [2]. The Event Calculus is introduced in detail in Section 2.1.
In their 2015 paper *An Event Calculus for Event Recognition* [3], Artikis et al. expand upon the original work by Kowalski and Sergot. They set out to make the Event Calculus viable for Complex Event Recognition in real time on heterogeneous datasets and succeed in doing so by introducing a dialect of the Event Calculus which they call *Event Calculus for Run-Time Reasoning (RTEC)* and a corresponding Prolog based implementation. Among other things, RTEC enables set-like operations such as `union` and `complement` on time intervals, which increases efficiency and power of expression while reasoning over time frames, as opposed to time instants or collections thereof. The paper and the RTEC dialect serve as basis for further work, particularly in the area of CER on maritime datasets.
Patroumpas et al. use RTEC in their system for monitoring maritime activity which they use successfully to detect noteworthy behavior of sea vessels in real time [6]. Their work is founded on the previously summarized paper by Artikis et al. albeit taking the approach in a slightly different direction.
In their industry paper on maritime monitoring, Pitsikalis et al. reshape this approach using domain expert knowledge and show the ease of use of RTEC predicates to define maritime events and benchmark their system on two datasets. They show that their approach is capable of CER in real time [4].
Further work by Tsilionis et al. introduces an incremented version of RTEC called $RTEC_{inc}$ which achieves improved efficiency by avoiding duplicated computations present in RTEC and outperforms it both in theory and in benchmarking [14]. These last two papers both use a dataset prepared and published by Ray et al. for the datAcron project [9]. Our work is based upon this line of research and uses it as reference wherever possible.
An extension of the Event Calculus using Markov Logic Networks in order to deal with uncertainty has been presented by Skarlatidis et al. [15]. The approach therein presented introduces a probabilistic component and improves robustness but ultimately extends beyond the scope of this thesis.
Outside of the canon of work which is build upon the EC, there are two logic based approaches to CER which themselves serve as the basis for subsequent work. Ghallab presents the *Chronicle Recognition System* (CRS) [16] which is further developed by Dousson [17]. It focuses on temporal constraints which are represented as graph edges connecting events represented as notes. Anicic et al. present ETALIS, an "expressive language for specifying and combining complex events" [18]. Both CRS and ETALIS are formalizations for logic-based CER, alternative to RTEC but not based on the EC.
Furthermore Barbieri et al. present a continuous extension to SPARQL[1] named C-SPARQL [19]. This extension has been used and built upon by other work focusing on subjects like industrial automation [20] or social media analytics [21], among others. The approach taken by C-SPARQL and subsequent works represents an alternative approach viable to implement stream reasoning.

2.1 The Event Calculus

In their article *A Logic-based Calculus of Events*, published in 1986, Robert Kowalski and Marek Sergot state that at the time of writing it was "still widely believed that classical logic is not adequate for the representation of time" ([2], p. 1). Time is not a simple concept to reason about in any context. It can be defined as the continued and perceivable existence of configurations of reality transitioning into each other. In this context, the term *event* can be used loosely to describe that which causes a transition.

[1] SPARQL is a query language standardized by the World Wide Web consortium: https://www.w3.org/TR/sparql11-overview/

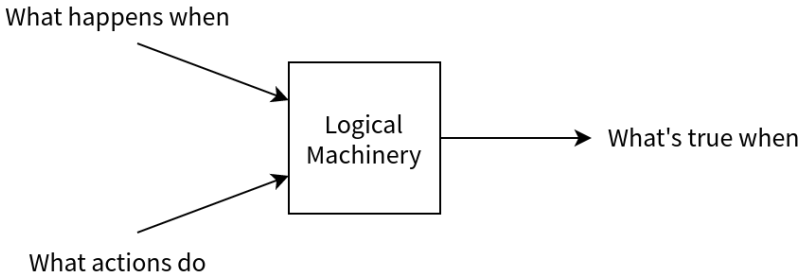

Figure 2.1.1: *What the Event Calculus does* [12]

A formalization of these concepts is required to achieve usefulness in a computational context, which is what the article presents ([2], p. 1):

> In this paper, we shall outline a treatment of time, based on the notion of event, formalized in the Horn clause subset of classical logic augmented with negation as failure. The resulting formalization is executable as a logic program.

The authors lay the foundation for the Event Calculus (EC) by defining an *event* as something which causes a transition (e.g. "Mary is hired as a lecturer", the implication being that she had not been a hired lecturer prior to the event) and which happens at a certain point in time (e.g. "May 10th 1970"), and a *predicate* as something which can *hold* or not, i.e. be true or not, at a certain point in time (e.g. "Mary is a hired lecturer"). Given only these two basic definitions, one can conclude that to ascertain whether a certain predicate holds, it is necessary for the most recent event which relates to the predicate to have resulted in the predicate being true, or in example form: Mary is a hired lecturer if and only if the most recent change to her rank made it so.

In his article *The Event Calculus explained* [12], Shanahan diagrams the basic mechanism of the Event Calculus as shown in figure 2.1.1. Shanahan argues that depending on which two of the three edges going into or coming out of the *Logical Machinery* block in the diagram are known and which of them is being inquired for, the task at hand is either *deductive* ("What happens when" and "What actions do" known, "What's true when" required), *abductive* ("What actions do" and "What's true when" known, "What happens when" required) or *inductive* ("What's true when" and "What happens when" given, "what actions do" required). In this classification, Complex Event Recognition (CER) falls into the category of deductive tasks, with the visualization by Shanahan providing also the explanation of how the Event Calculus can be used for CER.

The following practical example is presented to enable an intuitive understanding: a vehicle turning a corner is a complex event which can be described as a sequence of braking, then turning and finally accelerating again. The deductive task "Did the vehicle turn a corner?" decomposes into "Did the vehicle brake, then turn, and finally accelerate again?". This scenario can be formalized as an event which is recognized from a configuration of fluents. The formalization requires the following basic predicates:

- $\text{happensAt}(E, T)$ to denote that an event E was detected at time T
- $\text{initiatedAt}(F = V, T)$ to denote that a fluent F first took value V at time T
- $\text{terminatedAt}(F = V, T)$ to denote that a fluent F stopped having value V at time T

Using these predicates and the fluents $braking$, $turning$ and $accelerating$, the event $vehicleTurnedCorner$ (starting at time T_0) may now be stated as follows:

$$
\begin{aligned}
&\text{happensAt}(vehicleTurnedCorner(T_0)) \leftarrow \\
&\quad \text{initiatedAt}(braking,\ T_0), \\
&\quad \text{initiatedAt}(turning,\ T_1), \\
&\quad T_1 > T_0, \\
&\quad \text{terminatedAt}(braking,\ T_2), \\
&\quad T_2 \geq T_1, \\
&\quad \text{initiatedAt}(accelerating,\ T_2), \\
&\quad T_3 \geq T_2
\end{aligned}
\tag{2.1.1}
$$

This example is of course a very simple one. In order to be useful, it may be necessary to add the constraint that acceleration must not happen during braking, or constraints regarding time interval overlaps. It may also be more useful to express the inquiry as a fluent *vehicleTurningCorner* since it extends over a time interval itself. Using the limited set of EC predicates presented above this may quickly become cumbersome, which is why a more powerful *dialect* of EC formalization is needed.

2.2 A dialect of the Event Calculus for maritime monitoring

Since its inception, several dialects of Event Calculus formalizations have been developed, including those by Kowalski et al. in 1986 [2] and Shanahan in 2000 [12]. A more recent dialect presented by Artikis et al. in 2015 [3] is most relevant with regards to its applicability to CER. The authors present a dialect they call *Event Calculus for Run-Time Reasoning (RTEC)* which allows them to implement the Event Calculus as a CER software solution working on several heterogeneous datasets.

RTEC Among other things, RTEC introduces the notion of intervals to the predicate space, allowing for generative computation of lists of intervals and for set-like operations on them (e.g. complement, union, intersection, ...). The predicates of RTEC are listed in table 2.2.1.

Predicate	Meaning
happensAt(E, T)	Event E occurs at time T
holdsAt$(F = V, T)$	The value of fluent F is V at time T
holdsFor$(F = V, I)$	I is the list of maximal intervals for which $F = V$ holds continuously
initiatedAt$(F = V, T)$	At time T a period of time for which $F = V$ holds is initiated
terminatedAt$(F = V, T)$	At time T a period of time for which $F = V$ holds is terminated
relative_complement_all(I', L, I)	I is the list of maximal intervals produced by the relative complement of the list of maximal intervals I' with respect to every list of maximal intervals of list L
union_all(L, I)	I is the list of maximal intervals produced by the union of the lists of maximal intervals of list L
intersect_all(L, I)	I is the list of maximal intervals produced by the intersection of the lists of maximal intervals of list L

Table 2.2.1: Predicates of RTEC

These predicates — the interval manipulation constructs specifically — enable the construction of "simplified CE definitions and improve reasoning efficiency" ([3], p. 14). The implementation presented by Artikis et al. is done in Prolog and is shown by the authors to be viable for real-time CER using city transport management data as well as public space surveillance data. To achieve this, it incorporates the following techniques:

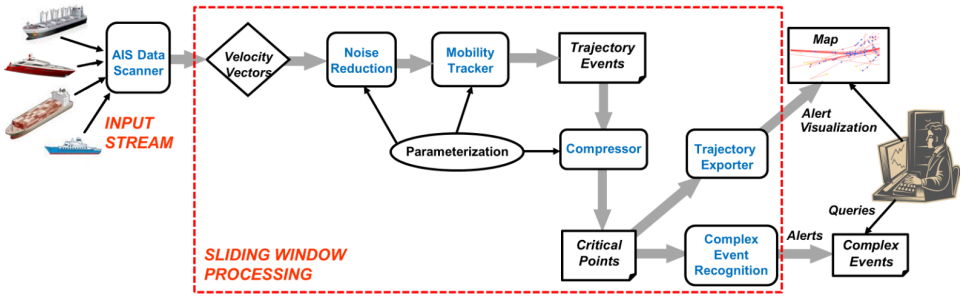

Figure 2.2.1: Maritime monitoring system architecture as presented by Patroumpas et al. [6]

- automated filtering out of irrelevant intermediate events
- cacheing: re-computations are avoided by storing intermediate events in memory
- windowing: CER is only performed on a time interval of predefined duration prior to the most recent time instant of all recorded data points, thus limiting the computational effort to events which can be recognized in this time interval

The windowing mechanism in particular represents a key feature of the presented implementation. At query time, the entire window is evaluated for occurring or continuing events, requiring careful calibration of the window width: a certain overlap of two subsequent windows is required but too much of an overlap and also a too large window width in general makes the evaluation too costly. However, the mechanism also compensates for data points being inserted *out-of-order*, i.e. not in the order of the time instants they were recorded at. In some data streams, this can occur for technical or other reasons and it is thus an advantage of a CER system if it implements robustness for this aspect. As such, the windowing mechanism is a feature which is both required for the computational efficiency of the presented RTEC implementation as well as a functional component of it.
Furthermore, the ability of the implementation to include both data streams and background knowledge in the evaluation is seen as a key strength of RTEC and therefore of the presented implementation. In the words of the authors ([3], p. 3):

> One of the main attractions of RTEC is that it makes available the power of logic programming to express complex temporal and atemporal constraints, as conditions in *initiatedAt* and *terminatedAt* rules for durative CEs, and *happensAt* rules for instantaneous CEs.

The combination of these aspects — (a) predicates capable of reasoning on both time instants and time intervals, (b) a windowing mechanism enabling both calibration for reduced computational cost and out-of-order data points and (c) the ability to easily integrate background knowledge — makes RTEC and its implementation viable for composing a CER system capable of real-time evaluation and of CE definitions which are "significantly more complex than those presented in most related papers" ([3], p. 14).

Maritime monitoring The term *maritime monitoring* describes the practice of using positional and other data transmitted by maritime vessels (i.e. ships sailing the ocean) to gain an understanding of their behaviour, often regarding regulatory compliance, in real time. Maritime vessels transmit such data using the *Automatic Identification System (AIS)* [22], a technology to provide identification, position and other information about vessels to other vessels and coastal authorities[2].
RTEC has been used for maritime monitoring applications in several papers. Patroumpas et al. present a monitoring system implementation which uses three main preprocessing stages — noise reduction, vessel trajectory pre-analysis and compression — before implementing CER using RTEC [6]. The architecture is shown in Figure 2.2.1. The authors present definitions of CEs using domain expert knowledge from the AMINESS project [23]. The dataset used for experiments in this work consists of AIS messages recorded

[2]Providing AIS data is a requirement of the International Maritime Organization for vessels of certain types and sizes: https://www.imo.org/en/OurWork/Safety/Pages/AIS.aspx

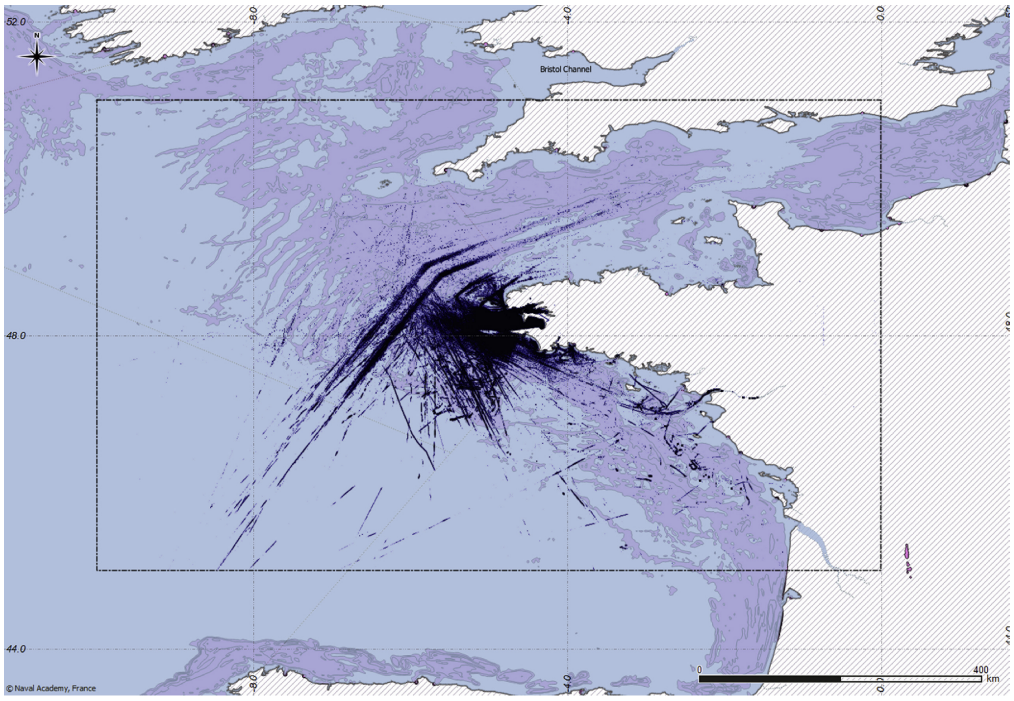

Figure 2.2.2: Navigation data [9]

from 1st June to 31st August 2009 from vessels in the Aegean, Ionian and part of the Mediterranean sea [24].
The work of Pitsikalis et al. builds on the paper by Patroumpas et al. and makes use of the same system architecture [4]. It uses domain expert knowledge from members of the French Naval Academy Research Institute (Brest, France) and the NATO Centre for Maritime Research and Experimentation (La Spezia, Italy). The datasets used in this work originate from the datAcron project [25] and contain AIS messages from vessels sailing around the port of Brest, France, between October 2015 and March 2016 and in the European seas in January 2016.
A further paper by Santipantakis et al. [5] also makes use of components of the system introduced in the paper by Patroumpas et al. as well as the CE definitions from the same paper, albeit it in heavily reduced form. Their work uses RTEC for CER only and not for spacial calculations as the previous papers, instead placing "emphasis on the detection of spatial relations" and developing "a separate component for highly efficient spatio-temporal link discovery" ([5], p. 2). Santipantakis et al. use the dataset from the datAcron project which covers the Brest area to show viability of their approach.
The dataset of vessels sailing in the area around the port of Brest is publicly available. It has been presented by Ray et al. [9] and contains around 18.6 million positional data points recorded by 4842 ships. Figure 2.2.2 shows the bounding box of the covered area and the recorded ship positions.

Strengths and weaknesses Both Patroumpas et al. and Pitsikalis et al. find that the preprocessing stages implemented by the system used in both works works effectively to reduce computational cost while not impacting accuracy, enabling real-time CER using RTEC. They highlight the ability of the system to execute preprocessing in parallel without going into detail regarding a possible implementation. Furthermore, they emphasize the quality and wide applicability range of the presented CE definitions of maritime activity developed in cooperation with domain experts.
The respective authors also mention weaknesses of the implementation. The need for calibration of parameters such as window width (for both preprocessing and RTEC) is mentioned, as well as the process for creating CE definitions. The fact that RTEC is implemented in Prolog and therefore executed in the Prolog runtime, which is capable of running in a single thread of execution only, is not directly addressed by Patroumpas et al., but Pitsikalis et al. state that "implementing RTEC in Scala in order to pave the way for distributed CER" ([4], p. 12).

2.3 Approaches for implementing concurrency in stream reasoning

In Chapter 2.4 *Operator Parallelization Methods* of their paper "A Comprehensive Survey on Parallelization and Elasticity in Stream Processing", Röger and Mayer present a set of methods for stream processing operator parallelization [26]. This set of methods identifies the different approaches for partitioning operations on data streams based on the specific tasks performed by the operations or based on the inherent characteristics of the data streams themselves, the latter of which may be conducted in several ways. The set of methods is summarized as follows:

- **Task parallelization.** Operations are applied to the same input stream in parallel by *multiplying*, i.e. making copies of, the input stream and independently running operations on the copies.
- **Data parallelization.** The input stream is partitioned into sub-streams using one of several methods and operations are applied to the sub-streams independently. The partitioning methods listed by the authors are the following:
  - *Shuffle grouping.* Data points are distributed independently to elements which apply processing operations using a scheduling algorithm such as Round-Robin.
  - *Key-based splitting.* Data points are grouped by an identifier (or "key"), resulting in several sub-streams to which operations are applied independently.
  - *Window-based splitting.* The input stream is partitioned based on time, splitting it into chunks related to a time interval (or "window"). Operations are applied to these chunks independently.
  - *Pane-based splitting.* With this method, a stream which is split based on windows is further split into smaller time based chunks (or "panes") to deal with overhead caused by overlaps.

In the canon of work based on RTEC, concurrency is achieved in some applications by running RTEC on several processor cores.
Artikis et al. [3] use variants on the above outlined methods for data parallelization for their experiments. When recognizing traffic events on data streamed from public transportation vehicles, the full input stream is sent to all RTEC instances with each instance only processing the sub-streams associated with the vehicles assigned to the respective instance, which is a variant on key-based splitting. When recognizing behavioral events of pairs of persons or objects (denoted *entities*), sequences of the input stream are processed in full by the RTEC instances, which is a variant on window-based splitting.
Both Patroumpas et al. [6] and Santipantakis et al. [5] achieve parallel execution by running several instances of RTEC which process data points from vessels sailing in geographically distinct sub-areas and handling non-concurrent scenarios occurring at the borders between sub-areas via specific implementation. This constitutes a variant on key-based splitting. While enabling parallel execution, this requires the additional step of implementing the splitting of data points by geographical sub-area and the handling of borders between sub-areas. Furthermore, concurrency is mentioned by the authors with regard to preprocessing stages [6] or the discovery of topological relations [5].
RTEC itself is run in various Prolog runtimes, none of which natively enable concurrency. No mention is made of running RTEC concurrently. It is therefore assumed that RTEC does not enable concurrency in its presented implementation.
In Chapter 3.1 of their survey, Röger and Mayer provide an overview of stream processing implementations [26], listing the operator parallelization methods each implementation makes use of or offers to its users. Their overview includes popular open source and commercial systems such as Esper [27] and projects of the Apache Foundation, namely Heron [28, 29], Storm [30], Spark [31] and Flink [32], which all provide some combinations of the parallelization methods outlined above. Furthermore the authors list scientific publications concerned with stream processing and CER and evaluate the respective approaches to parallelization of these publications. Table 1 of their work summarizes and lists all implementations and their properties, including but not limited to the parallelization methods they enable.
Overall the implementations presented by Röger and Mayer all focus on parallelism or distribution, not on a concurrency model for CER or logic-based stream reasoning. Many of them are purely declarative systems or query languages with limited capabilities to reason over intervals or with high degrees of complexity when it comes to formalizing processing tasks.

3 A formalization of the Event Calculus for enabling concurrency: ConEC

The purpose of any Complex Event Recognition (CER) system is to recognize events by processing data streams, thereby generating information that is consumed by an entity which uses said information to enable decision making. Consuming entities often set deadlines after which the usefulness of recognized events is diminished or eliminated, requiring CER systems to operate in real time. Most publications and specification of systems presented in Chapter 2 make specific mention of this capability.
RTEC, the Prolog implementation of an EC dialect presented by Artikis et al. [3], is no exception. All publications using RTEC in their respective implementations present experimental results showing that RTEC is capable of real-time event recognition [3, 6, 4]. In an experiment using traffic monitoring presented by Artikis et al. [3], RTEC performs CER on a real heterogeneous dataset using a window size of 30 minutes (corresponding to approximately 100 000 data points) in about 0.5 seconds. In the maritime monitoring application presented by Pitsikalis et al., the authors note ([4], p. 11):

> For example, in the Brest dataset, a window of 16 hours, including more than 50K events in the critical point stream, is processed in less than a second in a single core of a standard desktop computer.

Such results may put in question efforts to seek improvements in processing efficiency of CER systems. However, these results are achieved by exploiting application specific opportunities: the frequency with which maritime vessels send data points is low, the data stream is heavily preprocessed and processing times in the range of seconds (or minutes, as is the case in another experiment presented by Pitsikalis et al.) are viable for real-time operation. Furthermore, splitting data streams into overlapping geographic sub-areas to enable parallel execution of RTEC requires additional effort and computational overhead.
The formalization of the EC that is presented in this work considers fluents in terms of their dependencies, thus allowing for the identification of concurrency opportunities in the evaluation of functions produce fluents as their output. The need for application-specific implementations of pre- and post-processing mechanisms is therefore eliminated. Furthermore, this enables delegating the responsibility of efficient evaluation scheduling to a runtime which manages concurrent execution.
This chapter lists a set of methods for implementing concurrency which is based on the work by Röger and Mayer [26] outlined in Section 2.3 and discusses the challenges and opportunities for each method. Subsequently a formalization of the EC which enables a model for concurrency is presented and positioned as the basis for a prototypical implementation.

3.1 Methods for concurrent reasoning over data streams

Discussing concurrency in the context of stream reasoning requires the definition of concurrency by Lamport [11] to be extended from events to *predicates*, i.e. functions which are evaluated to obtain fluent values. Let f_a, f_b and f_c be such functions. Concurrency for these functions is defined as follows:

> If $f_a \rightarrow f_b$ then the output of f_a is an input to f_b and the two operations are not concurrent, meaning that f_a must be run to completion before f_b can start to be executed.
>
> Otherwise, if $f_a \nrightarrow f_b$ then the two fluent functions have no relation to each other and are concurrent, meaning that they can be executed arbitrarily.
>
> It follows that if $f_a \nrightarrow f_b$ and $f_b \nrightarrow f_c$ then also $f_a \nrightarrow f_c$.

Since CEs are represented as fluents which are the output of functions expressed as propositional predicate clauses, this extension applies to them.

In their survey on parallelization and elasticity in stream processing, Röger and Mayer categorize implementations to parallelize stream processing [26]. The parallelization methods they use in their categorization can be rephrased using the notation and terminology used in this work and defined in terms of concurrency models.

Task based concurrency CER systems which employ the task based concurrency model multiply the incoming data stream and process the resulting stream copies separately. Each of the stream copies is used as input to the fluent functions known to the CER system. For task based concurrency to be applicable, the fluent functions receiving stream copies must be concurrent.

Data based concurrency In CER system using the data based concurrency model, the input data stream is separated into sub-streams which are then processed separately. Each sub-stream is used to update the same set of fluent functions. For data based concurrency to be applicable, it must be possible to split the data stream into sub-streams on which the fluent functions of the CER system are concurrent. This work identifies three splitting strategies for obtaining sub-streams from the input data stream:

- *Shuffle grouping.* If the fluent functions of the CER system do not rely on stored state and are fully concurrent, each data point can be processed independently by any instance of the fluent functions, i.e. data points can be passed to any fluent function instance that is not currently busy.
- *Key based.* If the input data stream consists of several sub-streams with their own respective identifiers, sub-streams can be evaluated independently for fluent functions which are concurrent with respect to the sub-stream they are being evaluated on. For example, if the input stream from a collection of vehicles carries a sub-stream for each vehicle, those fluent functions concerned with single vehicles can be evaluated concurrently.
- *Window based.* In scenarios where fluent evaluation on data streams relies on stored state, it may still be possible to use windowing mechanisms. Such mechanisms partition the data stream into windows in the time domain and allow for concurrent fluent function evaluation on these windows.

Since different applications have differing characteristics in terms of their fluent functions and also in terms of their data streams, it is necessary that a concurrency model for the EC makes use of several if not all of these methods. Each of them presents challenges and opportunities when using them in a CER system.

Challenges and opportunities Röger and Mayer discuss the characteristics and limitations of the parallelization methods they present in their survey paper [26]. The following sections draw from their work, adapting and extending it where necessary.
Task based concurrency enables concurrent fluent functions to be evaluated independently. Because this is achieved via multiplying the input stream, it may lead to high resource demand for the multiplication and forwarding of data points. Depending on the implementation, the time variation required for the evaluation of different fluent functions and combinations of fluent functions may also be problematic.
Data based concurrency allows for separate sub-streams of an input data stream to be processed separately. This method is particularly efficient when using the shuffle grouping splitting strategy since scheduling can be based on load; however, it requires fluent functions to be fully concurrent. When using the key based splitting strategy, data based concurrency works well for sub-streams that have similar volumes; if not, it may lead to issues with balancing load. Window based splitting has the advantage of being configurable in terms of window size and overlap but may introduce unwanted management overhead due to application characteristics or inadequate configuration.
Combining these methods makes it possible to benefit from the opportunities while limiting the effect of the challenges, in particular in combination with flexible concurrency implementations which allow for processor-independent scheduling of concurrent operations. When fluent functions of the CER system can be evaluated concurrently across concurrency methods and splitting strategies, it is possible for the concurrency model to combine them freely. For example, if a data stream consists of several identifiable sub-streams on which the same fluents and CEs must be evaluated, it is possible to apply key based splitting for partitioning the input data stream into sub-streams and subsequently apply task based concurrency to evaluate each sub-stream.

3.2 Limitations of RTEC regarding concurrency

In RTEC, the concept of time intervals is used to reason over data streams. A tuple of recent data points from the input data stream — the *window* — is analyzed to find time instants in which fluents are initiated and terminated, thereby determining time intervals in which fluents *hold*. According to Artikis et al., this is "an implementation of the law of inertia" ([3], p. 2). CEs, which are represented by fluents, are recognized using interval operations on these time intervals. The following example demonstrates the manner in which RTEC is used by Pitsikalis et al. to recognize a maritime vessel exceeding a speed limit near a coast (simplified version of example presented by Pitsikalis et al. [4], p. 5):

$$
\begin{aligned}
&\text{initiatedAt}(highSpeedNearCoast(Vessel) = true,\ T) \leftarrow \\
&\quad \text{holdsAt}(withinArea(Vessel,\ nearCoast) = true,\ T), \\
&\quad threshold(v_{hs},\ V_{hs}),\ Speed > V_{hs}. \\
&\text{terminatedAt}(highSpeedNearCoast(Vessel) = true,\ T) \leftarrow \\
&\quad \text{happensAt}(\text{end}(withinArea(Vessel,\ nearCoast) = true),\ T) \\
&\text{terminatedAt}(highSpeedNearCoast(Vessel) = true,\ T) \leftarrow \\
&\quad threshold(v_{hs},\ V_{hs}),\ Speed \leq V_{hs}.
\end{aligned}
\tag{3.2.1}
$$

This composite statement determines the initiation of $highSpeedNearCoast$ being true at time T by checking if the vessel is located near the coast and the speed of the vessel exceeds the threshold for a speed violation. It determines the end of $highSpeedNearCoast$ being true at time T either by checking that the vessel is no longer located near the coast for a speed violation or by checking that the vessel's speed is no longer exceeding the threshold. The outputs of RTEC evaluations are:

- time instants and lists of time instants where CEs occur and
- time intervals and lists of time intervals where fluents (and thus CEs) hold.

RTEC is implemented in the Prolog language and is therefore executed by a Prolog runtime. Artikis et al. and Pitsikalis et al. both use YAP Prolog ([3], p. 9 and [4], p. 10) whereas Patroumpas et al. use SWI Prolog ([6], p. 19). Both of these runtimes run in a single process.
With regards to the concurrency methods outlined in Section 3.1, RTEC is therefore limited. Since RTEC does not support concurrency due to its implementation in Prolog, the authors of aforementioned works run several instances of RTEC in parallel on different processor cores. In the presented applications, it is therefore not practical or possible to use task based concurrency or shuffle grouping; window based concurrency is eliminated since the operation of RTEC is non-concurrent for sequential windows. Key based concurrency is applied by the authors, incurring the cost of managing geographic areas and borders between them in the example of maritime monitoring.

3.3 The ConEC formalization

The formalization of the EC presented in this work, referred to as *the Concurrent Event Calculus (ConEC)*, does not evaluate windows, instead processing data points as they are provided by the stream, to find time instants in which fluents are initiated and terminated. CEs, which are a specific type of fluent, are recognized as the status of the underlying fluents is updated. Contrary to RTEC, recognition does not happen by *looking back on a predefined number of data points* but rather as data points arrive. Evaluation results are stored and time instances and time intervals of interest are then found by querying the results. In order to describe the ConEC, three core components are required:

1. Separation of operation in two stages: *processing* and *observation*.
2. Specification of three separate fluent types: *data fluents*, *value fluents* and *boolean fluents*.
3. Definition of the predicates of ConEC.

These components are detailed in the following paragraphs.

Separation of operation in two stages The operation of the ConEC is separated into two stages, the *processing stage* and the *observation stage*.
In the processing stage, the data stream is evaluated with every new data point that arrives, updating all fluents accordingly. A stream of events, the Event Stream (ES), is emitted. The ES can either be consumed directly or persistently stored for later consumption by the observation stage.
In the observation stage, the ES is queried to find time instants and time intervals for which fluents have a certain value. However, contrary to RTEC, the ConEC makes it possible to query for the value of a fluent at a given time instant or time interval.

Specification of fluent types Recalling the definition from Section 1.3, a fluent is any property that varies over time, which can be a value from a data point delivered by the data stream or the output of a function which takes one or more fluents as its input. The latter can be distinguished further in terms of the type of produced output, which can either be a numeric or other application specific value or a boolean value. This also determines whether the fluent is expressed as a generic function or a propositional predicate clause. Since CEs are commonly represented by fluents which are boolean in their value type, the ConEC makes no distinction between CEs and such fluents.
It follows that there are three fluent types used by the ConEC:

- *Data Fluent (DF).* Fluents which are received as values from the data stream.
- *Value Fluent (VF).* Fluents which have a numeric or other application specific value.
- *Boolean Fluent (BF).* Fluents which have a boolean value, i.e. which can only be *true* or *false*.

This distinction is made in order to enable the definition of fluents in terms of their dependencies and thus enabling the concurrent evaluation of their respective fluent functions. The generic dependency relationship between the three fluent types is as follows.

- DFs are obtained directly from the data stream and have no associated fluent function.
- VFs have associated fluent functions which can have DFs, VFs and background knowledge as input, depending on them for evaluation.
- BFs have associated fluent functions which can have DFs, VF, other BFs and background knowledge as input, depending on them for evaluation.

All fluents are stored in memory and in the ES with their value and the timestamp of their latest update. BFs are stored additionally with the timestamp of the assumption of their current value. In this way, BFs allow ConEC to model time interval functions in CE definitions.

The predicates of the ConEC The predicates used in the ConEC are reduced compared to RTEC since there are no interval operations required in the processing stage, whereas any suitable querying mechanism can be employed in the observation stage. An implementation of ConEC will use the predicates to generate the ES and write it to a Structured Query Language (SQL) database, allowing for queries to be written in SQL. Table 3.3.1 lists the predicates of the ConEC.

Predicate	Meaning
$\text{valueAt}(F_{D\|V}, [K], T)$	Value of the fluent F of type DF or VF associated with the keys from list $[K]$ at time T
$\text{holdsAt}(F_B = V, [K], T)$	Fluent F of type BF associated with the keys from list $[K]$ has value V at time T
$\text{lastChange}(F_B, [K])$	Time instant of last change of fluent F of type BF associated with the keys from list $[K]$

Table 3.3.1: Predicates of the ConEC

Equation 3.2.1 shows the representation of a CE called $highSpeedNearCoast$ using RTEC. Equation 3.3.1 printed below shows the same CE represented using the ConEC. For the sake of presenting usage of all ConEC predicates, a change is applied to the CE $highSpeedNearCoast$ in that it is only recognized if the speed of a vessel exceeds the speed limit for at least 10 seconds.

$$
\begin{aligned}
&\text{holdsAt}(highSpeedNearCoast_B = true,\ [Vessel],\ T) \leftarrow \\
&\quad \text{holdsAt}(nearCoast_B = true,\ [Vessel],\ T), \\
&\quad \text{holdsAt}(highSpeed_B = true,\ [Vessel],\ T), \\
&\quad \text{lastChange}(highSpeed_B,\ [Vessel]) < (T - 10\,\text{s}). \\
&\text{holdsAt}(nearCoast_B = true,\ [Vessel],\ T) \leftarrow \\
&\quad \text{valueAt}(distanceFromCoast_V,\ [Vessel],\ T) \\
&\quad < nearCoastLimit_{BK}. \\
&\text{valueAt}(distanceFromCoast_V,\ [Vessel],\ T) \leftarrow \\
&\quad \text{valueAt}(location_D,\ [Vessel],\ T) \\
&\quad \text{operator}_{distance} \quad coast_{BK}. \\
&\text{holdsAt}(highSpeed_B = true,\ [Vessel],\ T) \leftarrow \\
&\quad \text{valueAt}(speed_D,\ [Vessel],\ T) \\
&\quad \geq highSpeedThreshold_{BK}
\end{aligned}
\tag{3.3.1}
$$

Consider the notation of fluents, which includes their type using subscript. In this example, the key is an identifier for a maritime vessel, denoted $Vessel$, and each list contains only one vessel. Items which are determined from background knowledge carry the subscript BK. The identifier "operator$_{distance}$" denotes a binary operator which calculates the distance between two objects in the location space of maritime monitoring.

3.4 Concurrency models enabled by the ConEC

Section 3.1 lists possible models of concurrency for CER systems. The ConEC enables all of the listed models and combinations of them. Applicability is determined only by the application specific representation of fluents using the ConEC predicates. In the following, the opportunities for concurrency models are outlined and exemplified.

Task based concurrency model Fluents which are concurrent with regard to their dependencies, i.e. fluents which have no dependency on each other or common non-DF dependencies, can be evaluated concurrently.
Consider the fluent $highSpeedNearCoast_B$ as given in Equation 3.3.1. Furthermore consider another fluent $searchAndRescueOperation_B$, the definition of which is omitted but shall be considered to not have a dependency on $highSpeedNearCoast_B$ nor on any of the non-DF dependencies of $highSpeedNearCoast_B$. For these two fluents, task based concurrency is applicable as they can be evaluated concurrently from copies of the data stream.

Data based concurrency models

- *Shuffle grouping.* Fluents which depend only on DFs or on background knowledge are viable for the shuffle grouping concurrency model.

  Consider the fluents $nearCoast_B$ and $highSpeed_B$ which are given in Equation 3.3.1 as dependencies of another fluent. Shuffle grouping is applicable as they can be evaluated concurrently and do not depend on prior information.

- *Key based concurrency.* In the ConEC, fluents are associated with key lists. Such fluents with a key list length of 1 or which have no keys in common can be evaluated concurrently.

  Consider a data stream containing data points for an arbitrary number of vessels. Such a data stream can be split into sub-streams with each sub-stream being associated with one vessel. The fluent $highSpeedNearCoast_B$ as given in Equation 3.3.1 can be evaluated for each sub-stream concurrently from all other sub-streams.

- *Window based concurrency.* The ConEC separates the processing and the observation stage. In the processing stage, window based concurrency is not possible nor required since evaluation happens continuously with every data point. In the observation stage, arbitrary window sizes can be chosen per CE definition (represented by a BF) to optimize queries.

  Consider the fluent $highSpeedNearCoast_B$ as given in Equation 3.3.1. The processing stage emits only the value of the fluent at each time instant into the ES, but does so with every data point that is received. It is up to the observation stage to find time intervals in which the fluent has a given value, or time instants when the fluent assumes a given value, or any such information which might be of interest to the CER system. In so doing, the observation stage may define windows of interest for each specific query.

The ConEC therefore enables the definition of fluents in a way which makes it straightforward to evaluate them concurrently. Dependencies which prevent concurrent evaluation are part of fluent definitions. Fluent types provide a clear separation between value properties and CEs. The inclusion of keys formalizes key-based concurrency. Lastly, the separation between processing stage and observation stage, with evaluation performed in the processing stage only, provides a more targeted application of window based concurrency.

3.5 A note on expressiveness and ergonomics

In the original work on the EC *A logic-based calculus of events*, Kowalski and Sergot state that they have "have potentially sacrificed the greater conciseness of modal logic for the greater expressiveness of an explicit treatment of time and events" ([2], p. 6). Artikis et al. mention in *An Event Calculus for Event Recognition* that "a major benefit of RTEC is that it supports the expression of rather complex definitions" ([3], p. 2). Further note of the expressiveness provided by certain implementations is made by Röger and Mayer in *A Comprehensive Survey on Parallelization and Elasticity in Stream Processing* [26]. As such, the ability to incorporate domain expert knowledge in a simple and expressive way is seen as a positive aspect of CER systems, and a strength of the EC and RTEC in particular. In this work, this aspect of a CER system is referred to as its *ergonomics*. Depending on implementation, the ConEC offers similar opportunities for ergonomic integration of domain expert knowledge as RTEC. Realizing these opportunities is therefore considered a main goal for the implementation of the ConEC.

4 Implementation of the ConEC

The Concurrent Event Calculus (ConEC) is a formalization of the EC for enabling concurrency with the goal of seeking improvements with regards to processing efficiency over previous implementations such as RTEC. The concurrent stream reasoning system based on the ConEC formalization presented in this chapter realizes the opportunities provided by the formalization presented in Chapter 3. The core elements of the ConEC formalization are defined in Section 3.3 as follows:

1. Separation of operation in two stages: *processing* and *observation*.
2. Specification of three separate fluent types: *data fluents*, *value fluents* and *boolean fluents*.
3. Definition of the predicates of ConEC.

The implementation is executed accordingly, with abstractions matching the formalization. In the processing stage in particular, computational efficiency and robust mechanisms for concurrent execution are required. Therefore the Rust programming language in conjunction with the tokio asynchronous runtime[1] were chosen as the implementation platform. Rust is a compiled, multi-paradigm programming language comparable to C/C++ regarding run time performance. The tokio runtime provides elements and tooling for implementing concurrently executing programs. Furthermore, the implementation makes extensive use of Rust language features to provide an ergonomic mechanism for implementing functions specified as ConEC predicates and thus integrating domain expert knowledge. This effort was made since the ergonomics of RTEC regarding the integration of domain expert knowledge are seen as one of its key strengths (as described in Chapter 3, Section 3.5).
The example application chosen is based on that which is presented by Pitsikalis et al. in their work *Composite Event Recognition for Maritime Monitoring* and is described in Section 4.1. Section 4.2 details the components of the implementation, focusing on the processing stage and the implemented variations of the observation stage. In Section 4.3, the concurrency models which have been realized in this implementation are discussed. Finally, Section 4.4 discusses the viability of the implementation for other applications.

4.1 Complex event recognition for maritime vessel monitoring

Maritime vessels, i.e. ships and boats sailing in the open seas or coastal areas, are subject to regulations which guide the operational behaviors in various aspects. For example, speed limits may be imposed near a coast or exchange of goods between vessels may be guided by local legislation, forcing vessels to attempt such exchanges in the open sea. As outlined in Chapter 2, Section 2.2, CER systems and RTEC in particular have been used by several related publications for monitoring maritime vessels and recognizing behavioral events which may be of interest to authorities.
The work by Pitsikalis et al. [4] serves as the basis for the use case for the implementation of the ConEC presented here. The authors present a set of complex event definitions developed with domain experts from the French Naval Academy Research Institute and the NATO Centre for Maritime Research and Experimentation. These definitions are implemented using RTEC and evaluated on a dataset provided by the datAcron project [25]. A subset of the definitions given by Pitsikalis et al. which is viable to show the implementation of concurrency is re-defined in this work using the ConEC.

The complex event definitions Pitsikalis et al. [4] define a catalogue of CEs recognizable on maritime vessel data, ranging from *Vessel within area of interest* and *Vessel with high speed near coast* to *Vessel*

[1] https://tokio.rs/

rendez-vous — and event which signifies two vessels meeting in the open seas, potentially to exchange critical goods — and *Search and rescue operations*. All CEs are described using the RTEC formalization. In this work, the CEs *Vessel with high speed near coast* and *Vessel rendez-vous* are chosen as these CEs and their components utilize all aspects of both RTEC and the ConEC. A modified version of *Vessel with high speed near coast* is shown in Section 3.3; Equation 4.1.1 shows the unmodified version with Equations 4.1.2 and 4.1.3 showing the CE's main components. The fluent $distanceFromCoast_V$ is a fluent representing the distance of a vessel from the coast.

$$\begin{aligned}&\text{holdsAt}(highSpeedNearCoast_B = true,\ [Vessel],\ T) \leftarrow\\ &\quad \text{holdsAt}(nearCoast_B = true,\ [Vessel],\ T),\\ &\quad \text{holdsAt}(highSpeed_B = true,\ [Vessel],\ T).\end{aligned} \tag{4.1.1}$$

$$\begin{aligned}&\text{holdsAt}(nearCoast_B = true,\ [Vessel],\ T) \leftarrow\\ &\quad \text{valueAt}(distanceFromCoast_V,\ [Vessel],\ T)\\ &\quad < nearCoastLimit_{BK}.\end{aligned} \tag{4.1.2}$$

$$\begin{aligned}&\text{holdsAt}(highSpeed_B = true,\ [Vessel],\ T) \leftarrow\\ &\quad \text{valueAt}(speed_D,\ [Vessel],\ T)\\ &\quad \geq highSpeedThreshold_{BK}\end{aligned} \tag{4.1.3}$$

The CE *Vessel rendez-vous* is shown in Equation 4.1.4 with Equations 4.1.5 and 4.1.6 showing the CEs main components. The fluent $rendezVousConditions_B$ shown in Equation 4.1.5 has further subcomponents which are omitted for brevity:

- The fluent $isTugOrPilot_B$ is a fluent that represents the type of the vessel as being either a *tug*, which is a type of vessel used to tug (or *tow*) other vessels, or a *pilot*, which is a type of vessel used to transport vessel pilots between vessels and the cost.
- The fluent $stoppedOrSlow_B$ is a fluent that, similarly to $highSpeed_B$ representing the vessel moving at an above-threshold speed, represents a vessel being stopped or moving at a below-threshold speed.
- The fluent $nearCoast_B$ is defined in Equation 4.1.2.
- The fluent $nearPorts_B$ is a fluent which, similarly to $nearCoast_B$ representing a vessel being located near a coast, represents a vessel being located near a port.

Notably, the fluent $proximity_B$ is a fluent which has multiple key dependencies, thus exemplifying a scenario in which key-based concurrency is not viable.

$$\begin{aligned}&\text{holdsAt}(rendezVous_B = true,\ [V_1, V_2],\ T) \leftarrow\\ &\quad \text{holdsAt}(rendezVousConditions_B = true,\ [V_1],\ T),\\ &\quad \text{holdsAt}(rendezVousConditions_B = true,\ [V_2],\ T),\\ &\quad \text{holdsAt}(proximity_B = true,\ [V_1, V_2],\ T),\\ &\quad \text{lastChange}(proximity_B,\ [V_1, V_2],\ T)\\ &\quad \leq (T - durationThreshold_{BK}).\end{aligned} \tag{4.1.4}$$

$$\begin{aligned}&\text{holdsAt}(\\ &\quad rendezVousConditions_B = true,\ [V],\ T\\ &)\leftarrow\\ &\quad \text{holdsAt}(isTugOrPilot_B = false,\ [V],\ T),\\ &\quad \text{holdsAt}(stoppedOrSlow_B = true,\ [V],\ T),\\ &\quad \text{holdsAt}(nearCoast_B = false,\ [V],\ T),\\ &\quad \text{holdsAt}(nearPorts_B = false,\ [V],\ T).\end{aligned} \tag{4.1.5}$$

$$\begin{aligned}&\text{holdsAt}(proximity_B = true,\ [V_1, V_2],\ T) \leftarrow\\ &\quad \text{valueAt}(distance_V,\ [V_1, V_2],\ T)\\ &\quad \leq proximityThreshold_{BK}.\\ &\text{valueAt}(distance_V,\ [V_1, V_2],\ T)\\ &\quad \text{valueAt}(location_D,\ [V_1],\ T)\\ &\quad \text{operator}_{distance}\\ &\quad \text{valueAt}(location_D,\ [V_2],\ T).\end{aligned} \tag{4.1.6}$$

The dataset The datAcron project provides a set of Automatic Identification System (AIS) data gathered in the geographic area around the port of Brest, France, from the 1st October 2015 to the 31st of March 2016 [25]. The data points of AIS contain a number of items, listed in the specification for AIS [22]. Relevant for the implementation presented here are the following items:

- a Unix timestamp, which is the number of seconds which have elapsed since 00:00:00 UTC on January 1st, 1970
- a vessel identifier
- the geographic longitude and latitude of the vessel's position
- the vessel's speed
- the vessel's type, which specifies if the vessel is a fishing vessel, a tug, a pilot vessel or one of several other types listed in the AIS specification [22]

The relevance of these items is derived from their use in the CE definitions provided in Paragraph 4.1. The dataset contains a total of 19 035 630 data points for the given time frame, which is equivalent to approximately 1.2 data points per second.

4.2 Implementation architecture and mode of operation

In the formalization of the ConEC, the operation performing CER is split in two stages: the processing stage and the observation stage. The implementation presented here therefore has two separate components which implement the respective stages as shown in Figure 4.2.1. The processing stage furthermore implements the fluent types while the observation stage, which consumes the event stream produced by the processing stage, requires only boolean fluents that represent recognized events. The predicates of the ConEC are represented by user-defined rules in the processing stage and by declarative queries in the observation stage.

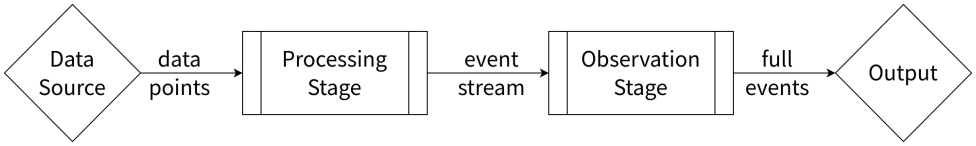

Figure 4.2.1: High level overview of the implementation

The processing stage The implementation of the ConEC presented here uses an architecture that relies on a broker service which manages the forwarding of evaluation results to the intended entities via a publisher-subscriber mechanism as described by Ozansoy et al. [33]. The implementation architecture is shown in Figure 4.2.2.

On startup, the *Broker* entity is configured with all its publishers and subscribers. The *Source* entity only publishes fluents; the *Sink* entity only subscribes to selected fluents which represent CEs. Each *FluentHandler* entity manages one fluent, subscribing to and collecting the dependencies for it and publishing it to the *Broker* entity after evaluating its predicate function. Exactly one *FluentHandler* is instantiated for each fluent.

In operation, the *Source* entity retrieves data points from the PostgreSQL database and separates them into fluents — each data point contains several items which are modeled as fluents in the ConEC definition, in this case $longitude_V$, $latitude_V$ and $speed_V$ — before publishing them to the *Broker* entity. The *Broker* entity forwards each fluent to all the subscribers of that fluent. The *FluentHandler* entities receive the fluents they are subscribed to and use them in their evaluation, which can require requests to the database to access background knowledge. The evaluation result is a fluent, which is published to the *Broker* and from there onwards to *FluentHandler*s which depend on it. In this way, dependencies between fluents are resolved: the respective *FluentHandler*s receive their dependencies as soon as they are available. Finally, the *Sink* entity receives the fluents it is subscribed to once their predicates have been fully evaluated, writing the resulting event stream back to the PostgreSQL database.

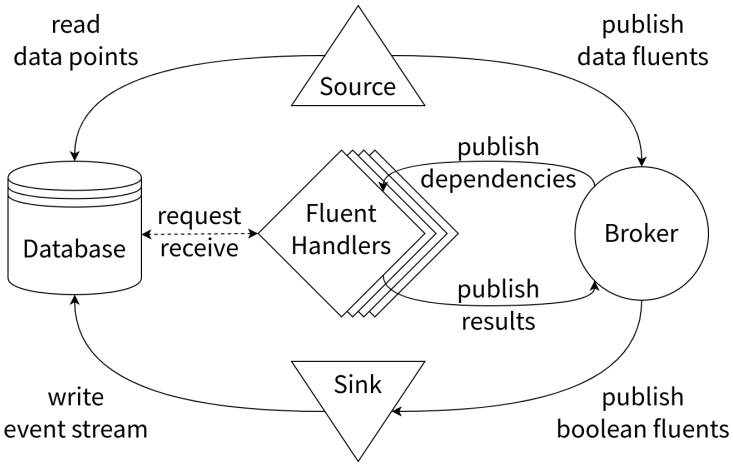

Figure 4.2.2: Architecture of the processing stage

The observation stage In this stage, the event stream emitted by the processing stage is evaluated. The event stream contains per-timestamp events. The purpose of the observation stage is to render the event stream usable to the final consuming entity, for example — to achieve equivalence with RTEC — by recognizing time intervals in which events occur or by visualization. The implementation presented here uses a PostgreSQL database to which the event stream is written, therefore the observation stage is implemented in PostgreSQL query language.
In this work, two variations on the implementation of the observation stage are presented. The first variation uses a PostgreSQL query to retrieve all for which fluents were continuously true. This enables the implementation to have an output that is semantically equivalent to that of RTEC.
Listing 4.1 shows a section of output from the implementation presented here. Every numbered row contains one interval in which a fluent holds the value `true` with four fields separated by commata. The first field contains the fluent name, the second field contains the list of IDs of associated vessels and fields three and four contain the starting and the ending timestamps respectively.
For reference, a section of raw output of the events detected by Pitsikalis et al. [4] using RTEC is shown in Listing 4.2. Each numbered line contains one interval in which a fluent holds the value `true` with the fields separated by the "|" character. The first field contains the fluent name, the second field contains the associated vessel ID, the third and fourth fields optionally contain information regarding potential dependencies and truth values and fields five and six contain the starting and the ending timestamps respectively.

```
1 "rendez_vous_conditions","{211232180}"
      ,1443679226,1443681205
2 "stopped_or_low_speed","{211232180}"
      ,1443679226,1443681205
3 "high_speed","{215872000}"
      ,1443679383,1443680482
4 "near_coast","{227002330}"
      ,1443679289,1443681270
5 "stopped_or_low_speed","{227002330}"
      ,1443679289,1443681270
6 "near_coast","{227003050}"
      ,1443679299,1443681279
```

Listing 4.1: Output using ConEC

```
1 stopped|245257000| |nearPorts
      |1443650414|1443650474
2 highSpeedNC|228131600| |true
      |1443650417|1443651361
3 withinArea|228931000|fishing|true
      |1443650439|1443651810
4 tuggingSpeed|228037600| |true
      |1443650440|1443661361
5 withinArea|234056000|nearPorts|true
      |1443650450|1443675363
6 lowSpeed|228037700| |true
      |1443650453|1443650501
```

Listing 4.2: Output using RTEC

The second implementation variation of the observation stage visualizes the event stream geographically. An PostgreSQL query is formulated which retrieves a predefined time frame while the implementation is running. Representative of how a CER system such as this implementation would be used, the time frame is chosen to include all events from the event stream which have a timestamp that is less then one hour older than that of the latest event in the stream. The result of the query is displayed on an

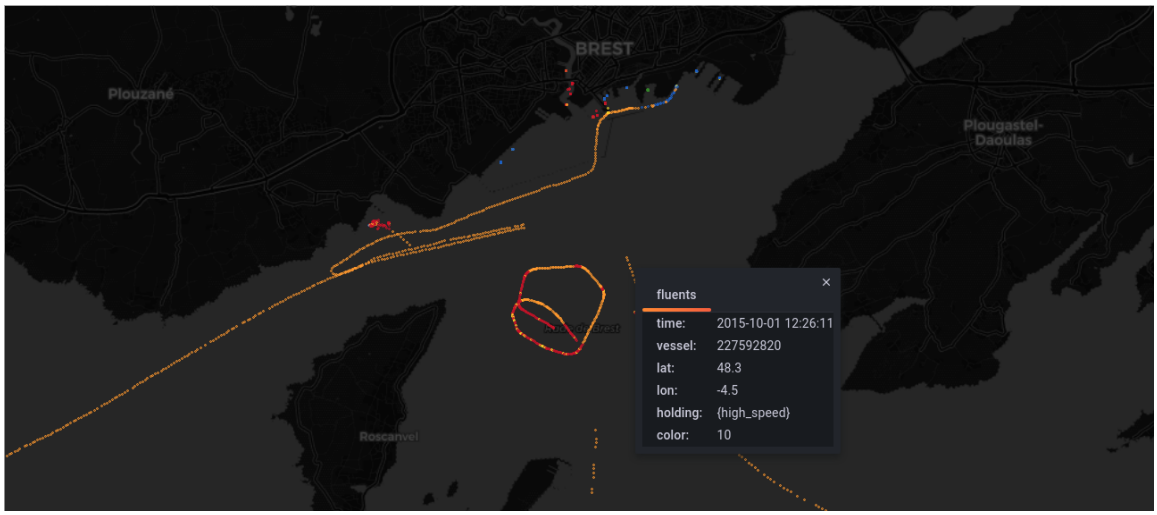

Figure 4.2.3: Geographic plot of fluent detected on a vessel

interactive map as shown in Figure 4.2.3. Each dot represents a fluent holding the value `true` for a vessel in the plotted location at a given time. The color of the dot encodes the type of fluent, e.g. orange for $highSpeed_B$, red for $stoppedOrSlow_B$ or blue for $nearCoast_B$.

The fluent types The formalization of the ConEC names three fluent types: Data Fluent (DF), Value Fluent (VF) and Boolean Fluent (BF). The implementation presented here utilizes a generic abstraction to represent all fluent types and specializes the abstraction for several Rust language basic or advanced types in the processing stage. In the observation stage, only BF are utilized and thus implemented via PostgreSQL `bool` type.

The predicates The ConEC predicates are implemented in the processing stage using a combination of the fluent types and Rust programming language functions. In addition to the name and value of the fluent, the fluent types contain the timestamp for which the fluent holds the value, associated key or keys of the fluent and an additional timestamp representing the *lastChange* predicate. The functions take fluents as their input and have fluents as their output.

4.3 Concurrency models realized in the implementation

Section 3.4 in Chapter 3 discusses the models of concurrency which are enabled by the ConEC formalization, i.e. which models of concurrency can be implemented when using the ConEC. The implementation of the ConEC presented here realizes all possible concurrency models.
Concurrency is implemented by defining units of program execution which independent from each other, referred to as *concurrent tasks*. Within a concurrent task, the tokio asynchronous runtime allows for program execution to be suspended while waiting for a resource or evaluation result to become available using the *await* keyword. The scheduler of the tokio runtime distributes tasks across available threads, executing the program until an *await* is encountered, at which point the scheduler will run again. The details of the scheduling mechanism used by the tokio runtime is beyond the scope of this work.

Task based concurrency model Each of the entities of the processing stage architecture shown in Figure 4.2.2 — *Source*, *Broker*, *Sink* and each of the *FluentHandler*s — are executed as concurrent tasks. This enables fluent predicates which are concurrent in their ConEC specification to be evaluated fully concurrently. Furthermore, data input, brokerage and output are also executed concurrently from fluent predicate evaluation. Lastly, the queries of the observation stage are evaluated independent from predicate evaluation. The task based concurrency model is thus fully implemented.

Data based concurrency models

- *Shuffle grouping.* The evaluation of fluent predicates within a *FluentHandler* is run within its own concurrent task for each data fluent input. Each task exists for as long as the evaluation runs and can be scheduled freely. While some non-concurrent overhead exists in the implementation, the basic mechanism is nonetheless one which enables concurrency by shuffle grouping at the task level.
- *Key based concurrency.* In the context of this use case, the keys are vessel IDs. Fluent predicates which have no dependency on fluents associated with more than one vessel are evaluated concurrently for their key via a shuffle grouped mechanism. Those predicates which do have a dependency on fluents associated with more than one vessel are evaluated concurrently for every viable combination of fluent dependencies based on vessel ID. Key based concurrency is therefore fully realized in this implementation.
- *Window based concurrency.* In the processing stage, time intervals are implemented via the *lastChange* parameter carried by the fluent type. However, the observation stage allows for windowed queries to optimize computational efficiency. Furthermore, processing and observation stage operate fully concurrently from each other. As the chosen implementation for the observation stage is PostgreSQL, a database system which allows for parallel queries, window based concurrency is fully realized by this implementation as well.

Summary The presented implementation realizes all concurrency models enabled by the ConEC. However, further opportunities for concurrency exist. For example, the architecture shown in Figure 4.2.2 could be extended to work with more than one *Broker* to allow for additional messaging concurrency. Furthermore, more *FluentHandler*s with key-based responsibilities could be added to allow for the concurrent execution of overhead functionality. These opportunities are left to be realized by future implementations.

4.4 Generalization of the implementation to other applications

The implementation of a CER system based on the ConEC which has been presented in this chapter has been executed for the application of maritime vessel monitoring. In its current form, it is viable and has been tested only for that application using the dataset provided by the datAcron project [25]. However, it is possible to use it for other CER applications as well.
If the CEs of an application can be specified using the ConEC formalization, only minor modifications are required. If furthermore the dataset for the application is available as a PostgreSQL database containing a data stream as specified in Chapter 1, Section 1.2, the implementation is fully viable as-is. However, given a application specific re-implementation of the *Source* entity shown in Figure 4.2.2, any set of data points expressible as ConEC data fluents can be used as an input to the implementation.
For the implementation of predicate functions, a basic understanding of the Rust programming language is required. Since ergonomics have been an aspect in the implementation design, the handling of this aspect does not require an in-depth understanding of concurrency.

5 Experimental evaluation

The implementation of a stream reasoning system presented in Chapter 4 enables concurrent Complex Event Recognition (CER) with the goal of making use of modern, multi-processor computing hardware. Concurrent execution of workloads has the potential of improving computational beyond what purely sequential execution can offer. Previous implementations such as RTEC are already capable of performing CER in real time for the maritime monitoring application based on the datAcron dataset which has been used also in this work. However, they do so by making use of extensive preprocessing and compression of the data stream as well as expensive mechanisms to enable multithreaded processing by running RTEC, an inherently sequential implementation of the EC, on multiple processing cores.

The concurrent stream reasoning system presented here does not make use of any such additional components, yet retains the ability to perform real-time CER for maritime vessel monitoring on the datAcron dataset. In fact, the experiments in this chapter show the limiting factor for the evaluation performance of the system is the response time of the PostgreSQL database, which is an order of magnitude higher than the time needed for predicate function evaluation. Furthermore, the experiments show that execution on multiple processing threads yields the expected benefits when compared to running the implementation on a single thread.

This chapter details the benchmarking environment of the experimentation and explains the components which have been selected for benchmarking. The experimentation results are then presented and subsequently discussed.

5.1 Description of the benchmarking environment

The experiments presented in this chapter have been conducted on a machine running the Debian 12 operating system using a Linux kernel at version 5.18.0. The processor installed in the machine was an AMD Ryzen 7 3800X with 8 physical cores, capable of executing 16 threads, operating at 4.25 GHz. The total memory capacity installed in the machine was 32 GiB, operating at 3600 MHz (DDR4). For compilation of the Rust program implementing the processing stage of the implementation, the `nightly-x86_64-unknown-linux-gnu` toolchain was used together with the `rustc 1.65.0-nightly` Rust compiler. The PostgreSQL database was used at version 11 running in a docker[1] environment.

Usage of operating system threads was controlled by functionality provided by the tokio asynchronous runtime, which allows for execution of the program with any number of *worker threads*. For benchmarking, all experiments were conducted using 1, 2, 4, 8 and 16 worker threads. Execution is scheduled concurrently irrespective of the number of worker threads by the tokio runtime.

At the time of experimentation, the machine was used for common user workloads as no dedicated machine was available. However, experiments showed the limiting factor to be database access — and this remained the case after refinements to the runtime configuration of the database aimed at improving response times — so that the use of a dedicated machine for experimentation was deemed unnecessary and no efforts were made to obtain access to a dedicated machine for experimentation.

5.2 Selection of benchmarks

The results presented in this chapter are obtained from selected benchmarking tests. The selection of the benchmarks is made in recognition of the composition of CE specifications, enabling measurements for subcomponents as well as full CE evaluations.

[1] https://www.docker.com/

Event	Description
$highSpeedNearCoast(V)$	Vessel V has high speed near coast
$anchoredOrMoored(V)$	Vessel V is anchored or moored
$drifting(V)$	Vessel V is drifting
$trawling(V)$	Vessel V is trawling
$tugging(V_1, V_2)$	Vessels V_1 and V_2 are engaged in tugging
$pilotBoarding(V_1, V_2)$	Vessels V_1 and V_2 are engaged in pilot boarding
$rendezVous(V_1, V_2)$	Vessels V_1 and V_2 are having a rendez-vous
$loitering(V)$	Vessel V is loitering
$sar(V)$	Vessel V is engaged in a search and rescue operation

Table 5.2.1: Composite events for maritime monitoring [4]

Cost of event recognition The cost of recognizing an event is the sum of the respective cost of all predicate functions which must be evaluated to recognize an event in the worst case scenario, equivalent to fully sequential program execution. In the best case scenario, i.e. when concurrent and fully parallel execution is realized, the cost of recognizing an event is that of the longest non-concurrent predicate function chain which must be evaluated.
The evaluation of predicate functions to recognize an event makes use of several mechanisms which have varying associated computational cost and therefore affect runtime behavior in varying ways. The mechanisms for predicate function evaluation are categorized as follows:

- inexpensive mathematical operations, e.g. comparisons to detect a value exceeding a threshold
- expensive mathematical operations, e.g. distance calculations
- evaluations which depend on fluent values from one vessel
- evaluations which depend on fluent values from more than one vessel
- operations requiring no database request
- operations requiring a database request once per vessel
- operations requiring a database request on each execution

Events selected for benchmarking The selection of benchmarks presented here is chosen in such a way that all mechanisms are covered and measurable. In their work *Composite Event Recognition for Maritime Monitoring*, Pitsikalis et al. [4] describe and — using RTEC — formalize a catalogue of events, most importantly nine *composite* events. These events are listed in Table 5.2.1. The specific meaning of the events in the context of maritime vessel operations is described in detail by Pitsikalis et al. [4] but is omitted here.
The implementation presented in Chapter 4 integrates the events $highSpeedNearCoast(V)$ as well as $rendezVous(V_1, V_2)$, with their specification using the ConEC formalization provided in Section 4.1. These events are chosen for benchmarking because they cover all mechanisms described above directly or through their components as follows:

- $highSpeedNearCoast_B$ depends on fluent values from one vessel only. It has the following contained components:
  - $nearCoast_B$ makes use of $distanceFromCoast_V$, which is evaluated using an operation that requires a database request on each execution. The component $nearCoast_B$ itself is then evaluated using an inexpensive mathematical operation in which the value of $distanceFromCoast_V$ is compared to the value of the background knowledge component $nearCoastLimit_{BK}$.
  - $highSpeed_B$ is evaluated using an inexpensive mathematical operation in which the value of the data fluent $speed_D$ is compared to the value of the background knowledge component $highSpeedThreshold_{BK}$.

Name	Value
$stoppedOrSlowLimit_{BK}$	5 kn
$highSpeedThreshold_{BK}$	5 kn
$nearCoastLimit_{BK}$	300 m
$nearPortsLimit_{BK}$	300 m
$proximityThreshold_{BK}$	100 m
$durationThreshold_{BK}$	30 min

Table 5.2.2: Values of background knowledge elements

- $rendezVous_B$ depends on fluent values from two vessels and is furthermore evaluated using an inexpensive mathematical operation in which the lastChange value of $proximity_B$ is compared to the value of the background knowledge component $durationThreshold_{BK}$ subtracted from the current time. $rendezVous_B$ has the following contained components:
  - $rendezVousConditions_B$ depends on values from one vessel only. Its contained components are $isTugOrPilot_B$, which requires a database request once per vessel; $stoppedOrSlow_B$, which is equivalent regarding its mechanism to $highSpeed_B$ described above; $nearCoast_B$, which is described above; and $nearPorts_B$, which is equivalent regarding its mechanism to $nearCoast_B$.
  - $proximity_B$ depends on values from two vessels. Its contained component $distance_V$ depends on values from two vessels and is evaluated using an expensive mathematical operation to calculate the distance between the vessels. $proximity_B$ is then evaluated using a an inexpensive mathematical operation in which the value of $distance_V$ is compared to the value of the background knowledge component $proximityThreshold_{BK}$.

Therefore the events $highSpeedNearCoast_B$ and $rendezVous_B$ and their contained components allow for measuring computational cost of all mechanisms and combinations of mechanisms. The values of the background knowledge components, insofar as not obtained from database requests, are listed in Table 5.2.2.

Description of benchmarking tests The implementation of the benchmarking tests presented here is executed via direct instrumentation of the processing stage. A monotonous timer is started by the *Source* entity and measurements are taken by *FluentHandler* entities which declare a dependency on the fluent for which they measure evaluation time. The measured evaluation duration is then written to a comma separated value file on the system running the benchmarking test. In this way, the duration required for evaluation of all fluents of interest from the point in time a data point enters the system to the point in time when the evaluation is complete is measured.
Evaluation duration measurements are taken for the following fluents:

- $highSpeedNearCoast_B$: composite CE
- $nearCoast_B$: inexpensive mathematical operation, depends on fluent values from one vessel, requires database request on each evaluation
- $highSpeed_B$: inexpensive mathematical operation, depends on fluent values from one vessel, requires no database request
- $rendezVous_B$: composite CE
- $isTugOrPilot_B$: depends on fluent values from one vessel, requires database request once per vessel
- $proximity_B$: expensive mathematical operation, depends on fluent values from two vessels

This selection of measurements covers all mechanisms and levels of complexity for the task of maritime vessel monitoring. All benchmarking measurements are taken with the full implementation running. The dataset used for the benchmarking tests is the datAcron dataset described in Chapter 4, Section 4.1, from which 26 733 data points recorded between 08:00 and 12:00 CEST, October 1st, 2015 are used.

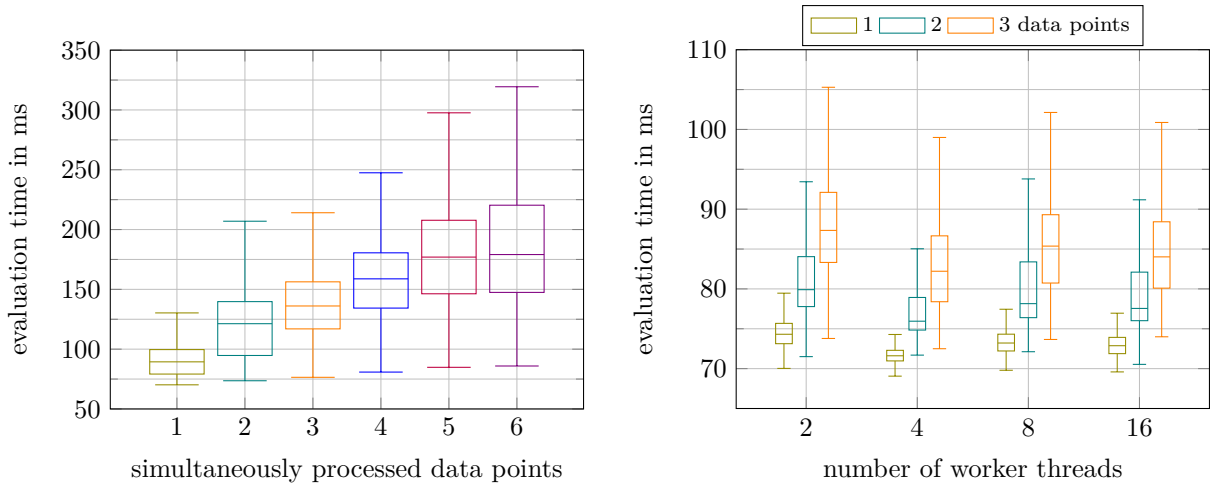

Figure 5.3.1: $highSpeedNearCoast_B$, single worker thread (left) and multiple worker threads (right)

5.3 Results

The benchmarking test results are presented in the order as listed in Section 5.2, Paragraph 5.2. The evaluation time measurements for the fluents $highSpeedNearCoast_B$, $nearCoast_B$, $highSpeed_B$, $rendezVous_B$ and $proximity_B$ are presented in Figures 5.3.1, 5.3.2, 5.3.3, 5.3.4 and 5.3.6 as box plots in the following configuration:

- The middle line represents the median.
- The lower and upper box edges represent the lower and upper quartiles respectively.
- The lower and upper whiskers represent the lowest and highest measurements within a distance of 1.5 times the interquartile range from the lower and upper quartile respectively.
- Outliers are omitted.

For each of the above listed fluents, a set of two plots is shown. The left side plot shows the evaluation time with the implementation running using one single worker thread in milliseconds on the vertical axis and the number of data points published by the *Source* entity in a single time step on the horizontal axis. The right side plot shows the evaluation time in milliseconds on the vertical axis and a group of three boxes for test runs with the implementation running using 2, 4, 8 and 16 worker threads. Each group represents the case of 1, 2 and 3 data points being published by the *Source* entity, from left to right.
The evaluation time measurement for the fluent $isTugOrPilot_B$ is presented in Figure 5.3.5 as a line plot. This plot shows the average evaluation time for the fluent when it is evaluated for a vessel for the first time and the average evaluation time for the fluent on subsequent evaluations for vessels, both in milliseconds, versus the number of worker threads on which the implementation is being run.

5.4 Discussion of results

The benchmarking test results presented in Section 5.3 first and foremost show the viability for real-time CER of the implemented concurrent stream reasoning system. All tests were run with a cycle time of 100 ms, with one cycle representing 1 s in the original data stream. Each test run covered a 4 h period and completed in 24 min. In the following, each test case is discussed separately.

High speed near coast Figure 5.3.1 shows that the total evaluation time for the CE $highSpeedNearCoast_B$ lies between under 100 ms for one data point published to about 175 ms for six data points published in the median when running the implementation on a single worker thread. These values drop to between

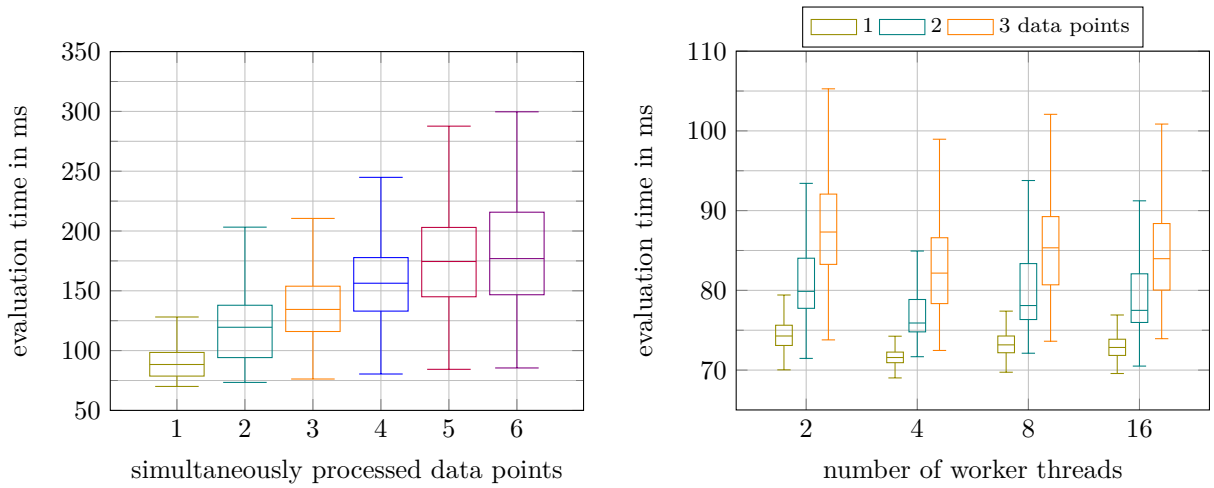

Figure 5.3.2: $nearCoast_B$, single worker thread (left) and multiple worker threads (right)

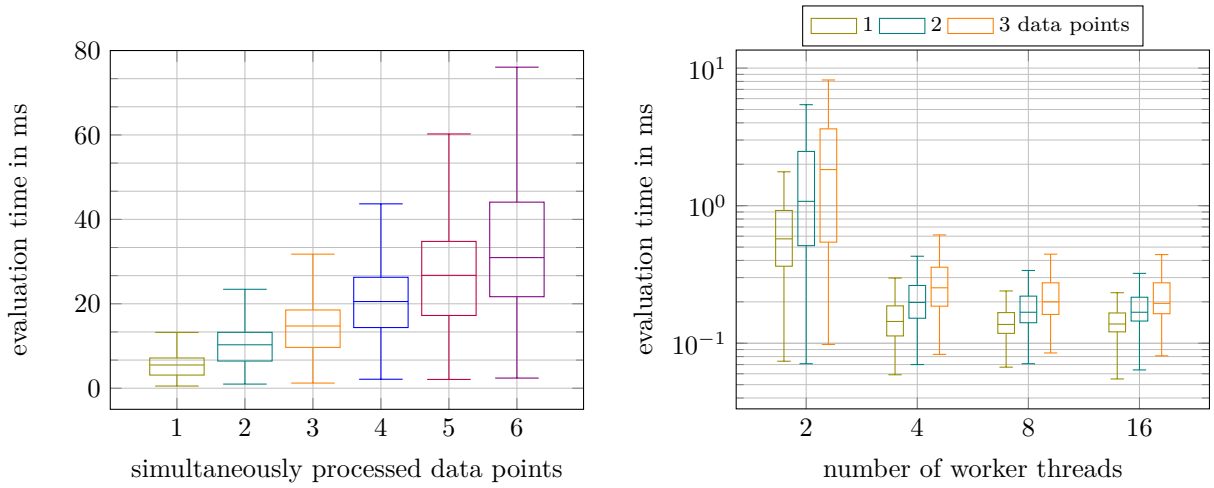

Figure 5.3.3: $highSpeed_B$, single worker thread (left) and multiple worker threads (right)

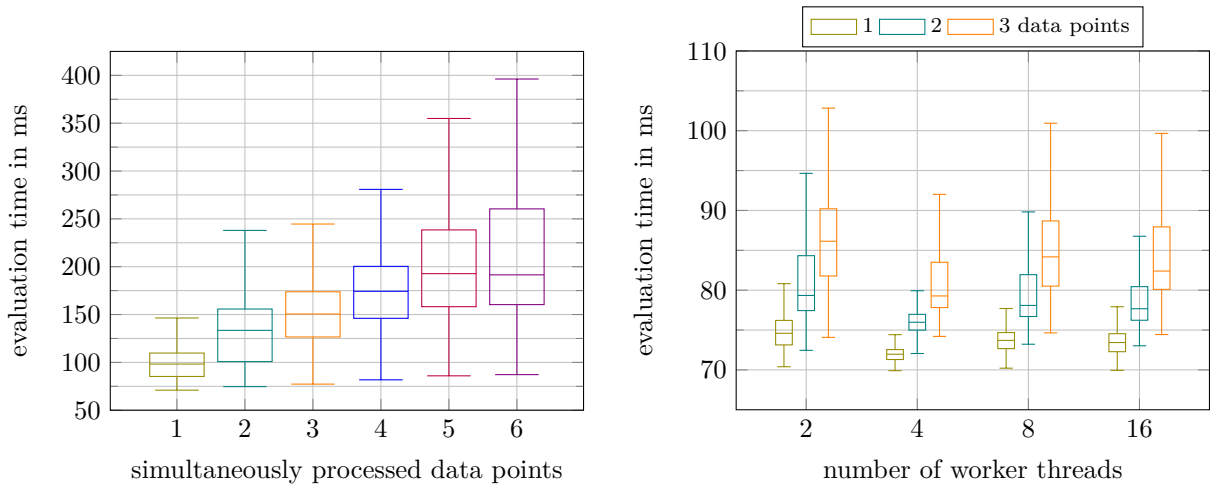

Figure 5.3.4: $rendezVous_B$, single worker thread (left) and multiple worker threads (right)

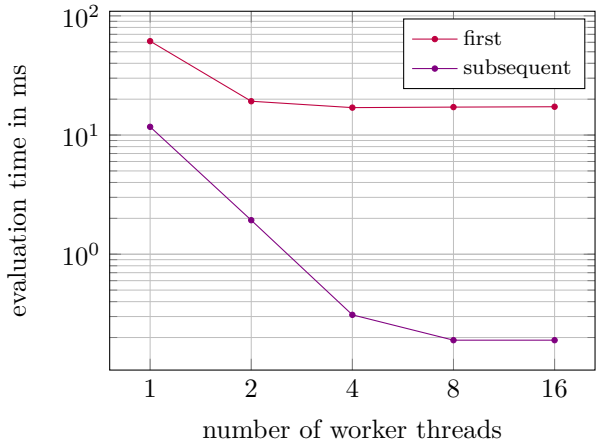

Figure 5.3.5: $isTugOrPilot_B$, average evaluation time

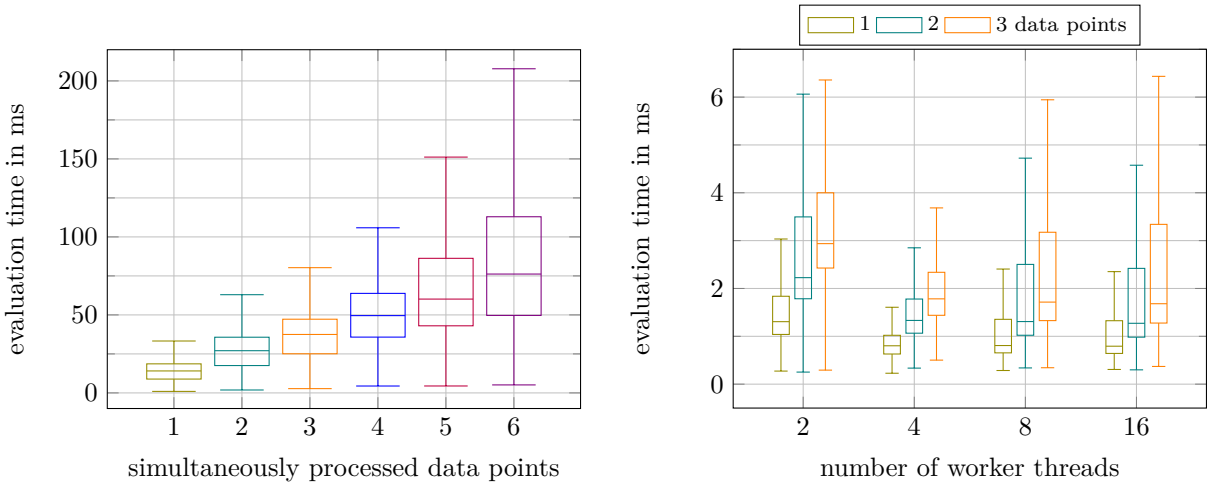

Figure 5.3.6: $proximity_B$, single worker thread (left) and multiple worker threads (right)

75 ms (one data point) and 87 ms (three data points) with two worker threads, and lower still with four worker threads. The measurements indicate, however, that the implementation does not benefit from more worker threads as the evaluation times increase slightly when measured using eight or 16 worker threads.
Figures 5.3.2 and 5.3.3 show evaluation times for the fluents $nearCoast_B$ and $highSpeed_B$ respectively. The measurements show that $nearCoast_B$ takes substantially longer in evaluation than $highSpeed_B$. It follows that evaluation times for $highSpeedNearCoast_B$ should be nearly identical with those of $nearCoast_B$, which is the case as shown by the presented measurements.
The fluent $nearCoast_B$ requires a database access on every evaluation, whereas $highSpeed_B$ is evaluated using an inexpensive mathematical operation only. The measurement results show that the evaluation times of $highSpeed_B$ are an order of magnitude shorter than those of $nearCoast_B$ regardless of data points published or worker threads utilized, indicating that database requests are indeed the limiting factor of the implementation. Furthermore, the results show that the benefit of utilizing more than one worker thread are more substantial for the evaluation of $highSpeed_B$ than for the evaluation of $nearCoast_B$. However, also the evaluation time of $highSpeed_B$ stagnates once four worker threads are utilized.

Vessel rendez-vous Figure 5.3.4 shows that the total evaluation time for the CE $rendezVous_B$ is approximately 100 ms for one data point published and close to 200 ms for six data points published in the median when a single worker thread is used. The benchmarking tests with more worker threads being utilized show evaluation times very similar to the evaluation $nearCoast_B$, which is a contained component of $rendezVous_B$ as well. This is indication that either the evaluation of $nearCoast_B$ or that of $nearPorts_B$ (which was not measured but is conceptually equivalent to $nearCoast_B$) are the determining factor for the evaluation time of $rendezVous_B$. Again, both of these fluent predicate functions require database access on each evaluation.
In Figure 5.3.5, the averages of evaluation time measurements for the fluent $isTugOrPilot_B$ are plotted. The predicate function for this fluent requires database access once for each vessel. The plot shows that the average time for the first evaluation for a vessel takes longer by an order of magnitude than subsequent evaluations, in which no database access is required. The measurements confirm that the benefits of utilizing more worker threads are more substantial when no database access is required. They also confirm stagnation of evaluation time benefits when utilizing eight or more worker threads.
The predicate function for the evaluation of the fluent $proximity_B$ depends on fluent values from two vessels as it recognizes two vessels being located in proximity of each other. It does not require database access but is evaluated using an expensive mathematical operation. Figure 5.3.6 shows evaluation time measurements for $proximity_B$. As shown, the evaluation time when utilizing one worker thread is similar but higher than that of $highSpeed_B$, which is within expectations. When utilizing more worker threads, the evaluation time is significantly reduced, reaching values of a few milliseconds which is equivalent to the evaluation of $highSpeed_B$. These results indicate that evaluations with a dependency on more than one fluent value benefit from concurrent execution. However, the previously observed stagnation at higher worker thread counts is confirmed by the presented measurements.

5.5 A note on the observation stage

In the presented implementation, the observation stage of the ConEC is implemented via PostgreSQL queries. The evaluation time of such queries depends on their complexity, which depends on their intended result, and on the size of the dataset they are run on. In manual testing, the query which has been implemented to realize an output equivalent to that of RTEC (see Chapter 4, Section 4.2, Paragraph 4.2) exhibited evaluation times of between 80 ms and 90 ms, consistent with the database access measurements conducted in the processing stage and presented in Section 5.4. However, no reliable experiments were conducted as the performance of PostgreSQL queries is not within the scope of this work.

6 Conclusion

This work presents the Concurrent Event Calculus (ConEC), a formalization of the Event Calculus which enables the specification of Complex Events for concurrent stream reasoning systems. Furthermore, a prototypical implementation of the ConEC is presented and shown to be viable for real-time CER on data streams.
As described in Chapter 1, this endeavour was motivated by the desire to explore the opportunities of concurrency — and thus the ability to make efficient use of modern computing hardware, which is increasing capable of multi-core processing — for logic based event recognition. The Event Calculus, outlined in Chapter 2, is a formalization for reasoning over events and was chosen as the foundation for this work because of its expressiveness and the extensive work by Artikis et al. on a dialect called Event Calculus for Run-Time Reasoning (RTEC). The core strengths of RTEC are considered to be its expressiveness and ergonomics and its implementation which is capable of performing real-time CER on data streams, as shown in numerous publications by its authors ([3, 6, 4] and others). In researching the subject, other approaches for implementing concurrency in stream reasoning were discovered and considered, yet the pure Event Calculus and RTEC served as the main inspiration for the ConEC.
In Chapter 3, the ConEC was formalized. The key target for the formalization was enabling concurrency with the goal of overcoming the limitations of RTEC in this regard while retaining its core strengths. A categorization of concurrency models is developed based on the work of Röger and Mayer [26]. The formalization of the ConEC provides a novel approach to the Event Calculus which incorporates concurrency in every aspect and which enables every concurrency model described in the categorization.
The prototypical implementation of the ConEC which has been produced for this work is presented in Chapter 4. It realizes the opportunities for concurrency via a publisher-subscriber architecture, which enables the resolution of non-concurrent dependencies in an elegant way. Furthermore, it strives to provide ergonomics comparable to RTEC.
Also described in Chapter 4 is the demonstration application which was used to develop and test the implementation: *maritime vessel monitoring* refers to the task of monitoring ships sailing the ocean with the goal of detecting potentially problematic behavior. This application is derived from previous work and a real-world dataset is publicly available for testing.
Lastly, the implementation of the ConEC is evaluated through benchmarking tests. The tests are derived from the demonstration application of maritime vessel monitoring. Chapter 5 catalogues a representative selection of test cases as well as their results. The viability of the implementation for real-time CER is shown as well as the benefits of concurrency with regard to computational efficiency: the implementation can process the test dataset an order of magnitude fast than required. However, the tests also bring the limitations of the implementation to light: database requests in particular play an important role, but also in cases where it is not the limiting factor, performance improvements stagnate at thread count number of more than four.

6.1 Future work

There are ample opportunities for future work regarding this work. The formalization of the ConEC is well defined regarding the component for logic based reasoning, but the component for rendering the generated event stream useful for users who might be domain experts from other fields should be refined. In the implementation itself, there are still opportunities for concurrency which should be evaluated with respect to their potential.
The architecture of the implementation should be further refined. In its current form, there exists a single choke point through which all data points and results must pass. Additionally, interfacing with the database has been shown in testing to be critical for performance. Lastly, further tests should also prove beneficial.

Glossary

AIS Automatic Identification System

BF Boolean Fluent

CE Complex Event

CER Complex Event Recognition

ConEC Concurrent Event Calculus

DF Data Fluent

EC Event Calculus

ES Event Stream

RTEC Event Calculus for Run-Time Reasoning

VF Value Fluent

SQL Structured Query Language

List of Figures

Bibliography

[1] Nikos Giatrakos et al. "Complex Event Recognition in the Big Data Era: A Survey". In: *The VLDB Journal* 29.1 (Jan. 2020), pp. 313–352. ISSN: 0949-877X. DOI: 10.1007/s00778-019-00557-w.

[2] Robert Kowalski and Marek Sergot. "A Logic-Based Calculus of Events". In: *New Generation Computing* 4.1 (Mar. 1, 1986), pp. 67–95. ISSN: 1882-7055. DOI: 10.1007/BF03037383.

[3] A. Artikis, M. Sergot, and G. Paliouras. "An Event Calculus for Event Recognition". In: *IEEE Transactions on Knowledge and Data Engineering* 27.4 (2015), pp. 895–908. DOI: 10.1109/TKDE.2014.2356476.

[4] Manolis Pitsikalis et al. "Composite Event Recognition for Maritime Monitoring". In: *CoRR* abs/1903.03078 (2019). DOI: 10.1145/3328905.3329762.

[5] Georgios M. Santipantakis et al. "A Stream Reasoning System for Maritime Monitoring". In: (2018). In collab. with Michael Wagner, 17 pages. DOI: 10.4230/LIPICS.TIME.2018.20.

[6] Kostas Patroumpas et al. "Online Event Recognition from Moving Vessel Trajectories". In: *GeoInformatica* 21.2 (Aug. 2016), pp. 389–427. ISSN: 1573-7624. DOI: 10.1007/s10707-016-0266-x.

[7] Pranav Tendulkar. "Mapping and Scheduling on Multi-core Processors Using SMT Solvers". In: (2014), p. 158.

[8] Alain Colmerauer and Philippe Roussel. "The Birth of Prolog". In: *ACM SIGPLAN Notices* 28.3 (Mar. 1993), pp. 37–52. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/155360.155362.

[9] Cyril Ray et al. "Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance". In: *Data in Brief* 25 (Aug. 2019), p. 104141. ISSN: 23523409. DOI: 10.1016/j.dib.2019.104141.

[10] Davide Barbieri et al. "Stream Reasoning : Where We Got So Far". In: (2010), p. 7.

[11] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359545.359563.

[12] Murray Shanahan. "The Event Calculus Explained". In: *In Artificial Intelligence LNAI* 1600 (June 2000). DOI: 10.1007/3-540-48317-9_17.

[13] Julian Clark. *Event Processing Glossary – Version 2.0*. Real Time Intelligence & Complex Event Processing. Aug. 23, 2011. URL: https://complexevents.com/2011/08/23/event-processing-glossary-version-2/ (visited on 03/30/2022).

[14] Efthimis Tsilionis, Alexander Artikis, and Georgios Paliouras. "Incremental Event Calculus for Run-Time Reasoning". In: *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems. Darmstadt Germany: ACM, June 24, 2019, pp. 79–90. ISBN: 978-1-4503-6794-3. DOI: 10.1145/3328905.3329504.

[15] Anastasios Skarlatidis et al. "Probabilistic Event Calculus for Event Recognition". In: *ACM Transactions on Computational Logic* 16.2 (Feb. 17, 2015), 11:1–11:37. ISSN: 1529-3785. DOI: 10.1145/2699916.

[16] Malik Ghallab. "On Chronicles: Representation, on-Line Recognition and Learning". In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*. KR'96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Nov. 5, 1996, pp. 597–606. ISBN: 978-1-55860-421-6.

[17] Christophe Dousson. "Extending and Unifying Chronicle Representation with Event Counters". In: *Proceedings of the 15th European Conference on Artificial Intelligence*. ECAI'02. NLD: IOS Press, July 21, 2002, pp. 257–261. ISBN: 978-1-58603-257-9.

[18] Darko Anicic et al. "ETALIS: Rule-Based Reasoning in Event Processing". In: *Reasoning in Event-Based Distributed Systems*. Ed. by Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa. Studies in Computational Intelligence. Berlin, Heidelberg: Springer, 2011, pp. 99–124. ISBN: 978-3-642-19724-6. DOI: 10.1007/978-3-642-19724-6_5.

[19] Davide Francesco Barbieri et al. "C-SPARQL: A CONTINUOUS QUERY LANGUAGE FOR RDF DATA STREAMS". In: *International Journal of Semantic Computing* 04.01 (Mar. 2010), pp. 3–25. ISSN: 1793-351X, 1793-7108. DOI: 10.1142/S1793351X10000936. URL: https://www.worldscientific.com/doi/abs/10.1142/S1793351X10000936 (visited on 07/04/2022).

[20] Franco Giustozzi, Julien Saunier, and Cecilia Zanni-Merk. "Abnormal Situations Interpretation in Industry 4.0 Using Stream Reasoning". In: *Procedia Computer Science* 159 (2019), pp. 620–629. ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.09.217.

[21] Davide Barbieri et al. "Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics". In: *IEEE Intelligent Systems* 25 (Nov. 2010), pp. 32–41. DOI: 10.1109/MIS.2010.142.

[22] *AIS Transponders*. URL: https://www.imo.org/en/OurWork/Safety/Pages/AIS.aspx (visited on 09/08/2022).

[23] *ANALYSIS OF MARINE INFORMATION FOR ENVIRONMENTALLY SAFE SHIPPING*. URL: https://www.iit.demokritos.gr/projects/aminess/ (visited on 07/25/2022).

[24] *Imis-3months | ChoroChronos.Datastories.Org*. URL: https://chorochronos.datastories.org/?q=content/imis-3months (visited on 07/25/2022).

[25] *datAcron H2020 ICT-16 Project*. URL: http://datacron-project.eu/ (visited on 07/25/2022).

[26] Henriette Röger and Ruben Mayer. "A Comprehensive Survey on Parallelization and Elasticity in Stream Processing". In: *ACM Computing Surveys* 52.2 (May 31, 2019), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3303849.

[27] *Complex Event Processing, Streaming Analytics, Streaming SQL*. EsperTech. URL: https://www.espertech.com/ (visited on 07/25/2022).

[28] *Apache Heron · A Realtime, Distributed, Fault-Tolerant Stream Processing Engine*. URL: https://heron.apache.org/ (visited on 07/25/2022).

[29] Sanjeev Kulkarni et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS'15: International Conference on Management of Data. Melbourne Victoria Australia: ACM, May 27, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788.

[30] *Apache Storm*. URL: https://storm.apache.org/ (visited on 07/25/2022).

[31] *Apache Spark™ - Unified Engine for Large-Scale Data Analytics*. URL: https://spark.apache.org/ (visited on 07/25/2022).

[32] *Apache Flink: Stateful Computations over Data Streams*. URL: https://flink.apache.org/ (visited on 07/25/2022).

[33] Cagil R. Ozansoy, Aladin Zayegh, and Akhtar Kalam. "The Real-Time Publisher/Subscriber Communication Model for Distributed Substation Systems". In: *IEEE Transactions on Power Delivery* 22.3 (July 2007), pp. 1411–1423. ISSN: 0885-8977. DOI: 10.1109/TPWRD.2007.893939.