



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**POSTGRADUATE STUDIES
“THEORETICAL COMPUTER SCIENCE”**

MASTER THESIS

**Complex Event Recognition: a comparison between
FlinkCEP and the Run-Time Event Calculus**

Alexandros-Nikolaos P. Troupiotis-Kapeliaris

**Supervisors: Panagiotis Stamatopoulos, Assistant Professor
Alexander Artikis, Assistant Professor**

ATHENS

November 2019



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
“ΘΕΩΡΗΤΙΚΗ ΠΛΗΡΟΦΟΡΙΚΗ”**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση Σύνθετων Γεγονότων: μια σύγκριση των FlinkCEP και Run-Time Event Calculus

Αλεξανδρος-Νικόλαος Π. Τρουπιώτης-Καπελιάρης

**Επιβλέποντες: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής
Αλέξανδρος Αρτίκης, Επίκουρος Καθηγητής**

ΑΘΗΝΑ

Νοέμβριος 2019

MASTER THESIS

Complex Event Recognition: a comparison between FlinkCEP and the Run-Time Event Calculus

Alexandros-Nikolaos P. Troupiotis-Kapeliaris

R.N.: M1610

SUPERVISORS: **Panagiotis Stamatopoulos**, Assistant Professor
Alexander Artikis, Assistant Professor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση Σύνθετων Γεγονότων: μια σύγκριση των FlinkCEP και Run-Time Event Calculus

Αλεξανδρος-Νικόλαος Π. Τρουπιώτης-Καπελιάρης

A.M.: M1610

ΕΠΙΒΛΕΠΟΝΤΕΣ: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής
Αλέξανδρος Αρτίκης, Επίκουρος Καθηγητής

ABSTRACT

The field of Complex Event Recognition (CER) on streams of data has shown remarkable growth the last few years. CER systems use streaming data in order to detect composite phenomena expressing relations between the input data. The amount of developed CER systems has created the need to examine and compare their capabilities. In this study we have chosen two systems, originating from the most dominant categories. From automata-based approaches we have selected FlinkCEP and from Logic-based systems we have selected RTEC. We present a theoretical comparison of the two systems' expressiveness, along with an empirical evaluation of the efficiency, using real data.

SUBJECT AREA: Logic Programming, Theory of Computation

KEYWORDS: event, recognition, complex events, automata, event calculus, data streaming

ΠΕΡΙΛΗΨΗ

Ο κλάδος της Αναγνώρισης Σύνθετων Γεγονότων πάνω σε ροές από δεδομένα έχει επιδείξει αξιοσημείωτη ανάπτυξη τα τελευταία χρόνια. Τα συστήματα αναγνώρισης σύνθετων γεγονότων περιεργάζονται ροές από δεδομένα με σκοπό τον εντοπισμό σύνθετων φαινομένων, που εκφράζουν σχέσεις ανάμεσα στα δεδομένα εισόδου. Ο αριθμός των συστημάτων που έχουν αναπτυχθεί τα τελευταία χρόνια έχει δημιουργήσει την ανάγκη για μελέτη και σύγκριση των δυνατοτήτων τους. Σε αυτήν την μελέτη επιλέγουμε δύο συστήματα από τις πιο επικρατούσες κατηγορίες. Διαλέγουμε το FlinkCEP από τα συστήματα βασισμένα σε αυτόματα και το RTEC από τα συστήματα που χρησιμοποιούν λογική. Παρουσιάζουμε μια θεωρητική σύγκριση της εκφραστικότητας των δύο συστημάτων, μαζί με μια πειραματική αξιολόγηση της αποδοτικότητας τους, χρησιμοποιώντας πραγματικά δεδομένα.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Λογικός Προγραμματισμός, Θεωρία Υπολογισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: αναγνώριση γεγονότων, σύνθετα γεγονότα, αυτόματα, λογισμός δράσης, ροή δεδομένων

ACKNOWLEDGEMENTS

At this point I should express my gratitude towards the people that played major roles during this study. Firstly, I would like to thank my advisors at NCSR Demokritos, Dr. Alexander Artikis and Dr. Georgios Paliouras for allowing me to be part of their group. Their trust and constant guidance are truly appreciated. Furthermore, I would like to thank Dr. Panagiotis Stamatopoulos, from University of Athens, for monitoring my work throughout its course.

Also, the advice of Elias Alevizos, Evangelos Michelioudakis and Manos Pitsikalis is appreciated to the fullest on both moral and technical dilemmas; their contributions to this thesis are of significant importance. A special reference to Christos Vlassopoulos for introducing me to the CER group and for supporting me, especially during the few first months. My gratitude to numerous researchers working on Demokritos, including but not limited to L.Tsekouras, M.Ntoulas, P.Mantenoglou, D.Kaklis, E.Tsilionis and N.Katzouris.

Last but not least, to my family for their continuous support in so many ways, and their patience over my constantly emerging concerns

CONTENTS

List of Figures	10
List of Tables	12
1 INTRODUCTION	13
1.1 Motivation	13
1.2 Contributions.	13
1.3 Outline of Thesis	14
2 RELATED WORK	15
2.1 Complex Event Processing and Recognition Systems	15
2.1.1 Automata-based Complex Event Recognition systems	16
2.1.2 Logic-based Complex Event Recognition systems	16
2.2 Comparisons	17
3 BACKGROUND	19
3.1 The FlinkCEP System	19
3.2 Event Calculus	24
3.2.1 The RTEC System.	24
4 THEORETICAL COMPARISON	27
4.1 Unbounded Intervals	27
4.2 Simultaneous initiation and termination	28
4.3 Simultaneous Events	29
4.4 Relations between patterns	32
4.5 Conclusions	35
5 EMPIRICAL COMPARISON	36
5.1 Experimental Setup	36
5.1.1 The Datasets	36
5.1.2 Complex Events and their Implementations	41
5.1.3 Comparison Criteria	47

5.2	Comparison	48
5.2.1	Maritime Dataset	48
5.2.2	Surveillance Dataset	52
5.3	Lessons Learned	54
6	CONCLUSIONS AND FURTHER WORK	55
6.1	Conclusions	55
6.2	Future work	56
	ACRONYMS	57
	REFERENCES	58

LIST OF FIGURES

Figure 1	RTEC is able to detect unbounded matches, in case the stream does not include an ending point, while FlinkCEP cannot simulate this behavior. .	27
Figure 2	While FlinkCEP would accept a match that is defined by simultaneous initiations and terminations, RTEC automatically filters such matches. In order to avoid these matches we should include an additional temporal condition.	28
Figure 3	The two streams include the same information but are treated differently by FlinkCEP . A pattern that requires for the (a) and (c) events to occur simultaneously requires a more complex approach for FlinkCEP	29
Figure 4	The two streams include the same information but are treated differently by FlinkCEP . A pattern that requires for the (a) and (c) events to occur simultaneously requires a more complex approach for FlinkCEP . The FlinkCEP implementation cannot detect the first match as no events are included into the stream in order to be used as bookmarks.	31
Figure 5	The prerequisite pattern (defined by (s) and (t)) should not hold during initiation of the pattern we are trying to detect. The first FlinkCEP approach would be to include the termination of the prerequisite as part of the pattern, missing the first match. The second, marks the first pattern's occurrence as optional allowing for the system to ignore it even if it appears.	34
Figure 6	The maritime patterns hierarchy, (after [39]). The patterns used for our evaluation are highlighted appropriately.	41
Figure 7	Two people meet / Meeting Context, (after CAVIAR's documentation ¹).	44
Figure 8	Total recognition time comparison of the two systems (with and without the use of windows in RTEC) for all maritime patterns for the full 6-month Maritime dataset. RTEC is able to detect all patterns passing through the dataset one. FlinkCEP requires separate CER for each pattern; we use the sum of all recognitions.	49
Figure 9	Recognition Time comparison of the two systems for each pattern for the full 6-month Maritime dataset using temporal windows in RTEC. . .	49
Figure 10	Recognition Time comparison of the two systems for each pattern for the full 6-month Maritime dataset without the use of temporal windows in RTEC.	50
Figure 11	Total execution time comparison of the two systems for each pattern for the full 6-month Maritime dataset.	50
Figure 12	Recognition time comparison of the two systems for each pattern for the 'meeting' pattern for the 1×CAVIAR and 10×CAVIAR datasets.	53

Figure 13 Total execution time comparison of the two systems for each pattern
for the 'meeting' pattern for the $1\times$ CAVIAR and $10\times$ CAVIAR datasets. . . . 53

LIST OF TABLES

Table 1	RTEC predicates and operators used for rules of Complex Events (after [6]).	25
Table 2	Temporal, spatial and entity attributes of Brest dataset.	37
Table 3	Table of Simple Derived Events appearing as input for the Maritime dataset, originating from Brest. All input events are instantaneous except 'proximity'. The first three types of events are a result of a spatial preprocessing; the next two originate directly from the AIS messages, while the rest are produced by the trajectory synopsis generator.	38
Table 4	Entity and SDE attributes of CAVIAR dataset.	39
Table 5	Table of Simple Derived Events appearing within the CAVIAR dataset, along with number of Ground Truth events.	40
Table 6	Maritime Accuracy Comparison. Comparing the results of both systems, supposing the RTEC results to be true, and using timepoints as a unit; hence the True Positives occur on both systems, the False Negatives only on the RTEC matches and vice versa for the False Positives.	48
Table 7	Similarity Comparison for Surveillance pattern. We evaluate the results of RTEC compared to FlinkCEP . In order to do so, we chose to use the RTEC implementation as Ground Truth and evaluate the FlinkCEP results correspondingly. We also are using the timepoints returned as units of our comparisons. The stream is being parsed into Keyed streams, for our FlinkCEP pattern to be simpler.	52
Table 8	Meeting Accuracy compared to the Ground Truth. We evaluate the results of the RTEC 's and the FlinkCEP 's implementations compared to the Ground Truth given. The GT corresponds to the 'meeting' value of the Context tag. We also are using the timepoints returned as units of our comparisons. The stream is being parsed into Keyed streams for our FlinkCEP pattern to be simple.	52

1. INTRODUCTION

In today's world the use of large data sources to extract useful information of a higher value than the data itself is a default component to most industries. Numerous projects include applications with sources that continuously feed the system with information. These applications range from handling messages arriving to a communications satellite, to detecting market trends on financial matters or even monitoring the movement of military operations. These data are included into streams and provided as separate packages of information (events) as time moves forward. Streaming applications are focused on forgetting already processed events.

Patterns between the included events can be detected on multiple occasions. These patterns may be simple sequences of events, the occurrences of singular events with certain characteristics or even more complex patterns. The detection of these patterns is a process that intends to find Complex Events [22].

1.1 Motivation

We examine two engines originating from the most dominant categories, automata-based and logic-based CER systems. A brief summary for each system is provided below, as well as a more descriptive presentation of their features throughout the rest of this thesis.

FlinkCEP Developed by Apache, the FlinkCEP system ¹ is built on top of one of the most widely used Streaming Platforms in the world, Flink, and thus it enjoys all its capabilities and its efficiency regarding handling big loads of data [11]. The patterns that can be expressed using FlinkCEP's dialect are based on an enhanced version of automata. They are se-

quences of events equipped with the filtering options and conditions that might concern a single or multiple of the match's components. Developed using the JVM capabilities and allowing the user to use both Java and Scala programming languages, FlinkCEP takes advantage of numerous features not available on other systems.

RTEC As an indicative system based on logic, RTEC : Event Calculus for Run-Time reasoning ², is based on the principles of Event Calculus [7]. It has been used for multiple fields, such as Maritime monitoring or camera surveillance applications. Capable of using temporal windows (partitions of the stream based on time), RTEC approaches the recognition process as the detection of the maximal intervals, defined by conditions on events occurring on the stream.

1.2 Contributions

The contributions of this study include:

¹<https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html>

²<https://github.com/aartikis/RTEC>

- A theoretical comparison of FlinkCEP and RTEC , focusing on their expressiveness.
- An empirical comparison in monitoring the movement of vessels at sea and detection of human interactions based on surveillance footage.

1.3 Outline of Thesis

The rest of the report consists of the following chapters: first (Chapter 2) we present related articles and papers that have investigated the capabilities and uses for several CER systems. Afterwards, (on Chapter 3) we present the two systems in question, along with few information regarding the recognition model they follow. Moreover, we theoretically compare the capabilities of the two systems by enlisting possible scenarios where a difference on the behavior of both systems can be spotted (Chapter 4). Moving on, we are describing the empirical evaluation we transacted; before providing the implementations of the patterns for those experiments and giving our remarks upon the results (Chapter 5). Finally, (Chapter 6) we summarized our conclusions, as well as mention several aspects of the comparison that could be expanded in future studies.

2. RELATED WORK

In this chapter we present previous work published that deals with streaming, describing the field of CER. Also we present the features of major CER engines, along with mentioning studies that compare such systems and their capabilities.

2.1 Complex Event Processing and Recognition Systems

As mentioned, the growth that the field of Data Science and Data Analysis has shown over the last years has been remarkable. The fact that the load of data is of a great scale in increasingly more applications creates a challenge. Handling it using traditional databases with few and infrequent insertions, seems to be an inefficient approach. As mentioned in [33], developing algorithms that handle data streams as their input, provides us with elegant solutions to problems such as sampling and extracting frequent items, similarity comparisons and summarizing.

Furthermore, techniques and architectures for performing Complex Event Processing (CEP) has been proposed by [25], as different algorithms are described. It has been suggested that these Complex Events can be expressed as queries on an Dynamic Query Evaluation Database System; though evidence has shown that major motivation exists in order to study and develop algorithms that concern queries that could be computationally lower when evaluated using CEP systems [47]. A more theoretical approach of CEP can be found in [29]; describing a formal CEP language and its operators, providing also examples of Complex Events and individual stream scenarios, without though providing an evaluation on real data, focusing on a more theoretical approach.

Several different implementations to the problem of CER have been developed. As mentioned by [28] a classification method for those types of approaches separates CER systems into three categories:

- The first, and most popular category, is comprised of systems based on automata. The patterns designed by the user are translated into an automaton. These resulting compiled automata follow rules of finite-state automata (FSA), but are also equipped with features regarding the attributes of input events as well as registers that allow storing information (usually previous events) during the recognition process. Furthermore, in this approach the concept of time is treated simply as an extra attribute of the input events. Several operators in our patterns, like optionally skipping events, may result in non-deterministic automata; hence the set of all potential matches could become exponential in the number of events being processed.
- The second is the category of engines that are built upon tree-based models, regarding both Complex Event modeling and recognition. Patterns designed by systems that fall into this category are modeled as trees of operators, such as sequence of events, with the input event usually appearing on leaf nodes. Moreover, tree based techniques are of paramount importance in several recognition algorithms, and combined with other techniques (like automata-based recognition) lead to hybrid approaches. Fewer systems have been developed using this approach, with the most prominent one probably being *Zstream* [37].

- The third category refers to logic-based engines. As with tree based models, these approaches can be applied on the modeling of our patterns and/or the detection methods. These systems tend to be more expressive and their modeling follows rigorous mathematical models. The recognition may be implemented using a logic programming language or be simulated by other types of approaches, such as automata or temporal constraint networks [8].

A question that arises at this point is how do systems from each category differ between them. Below we expand on several systems based on automata and logic before moving on several comparisons of CER systems that have been conducted.

2.1.1 Automata-based Complex Event Recognition systems

While taking all the attributes desired for a Complex Event language into consideration, the resemblance to automata theory and grammar comes as conspicuous to the reader. Features like the expression of the pattern as a sequence of input events or the need of iteration throughout the elements of the accepted pattern-match on the recognition process, come as the most significant example [28]. As a result, recognition based on automata appears to be probably the most dominant approach, as systems like *SASE*, *Caguya* or FlinkCEP tend to be substantially popular among commercial and academic applications.

Being one of the earliest examples of CER engines, *SASE* [30], played a major role on the rise of popularity of CER systems around the world. *SASE*'s language uses sequence based logic in order for the user to define Complex Events for recognition over a stream, translating them into (possibly non-deterministic) automata and performs a recognition process over the input stream. One of the most cited systems, *SASE* owes its popularity to the simplicity of its language and most precisely the similarity of its patterns to SQL queries.

Several systems that have similar design principles with *SASE* are available for users to experiment. Similar to *SASE*'s pattern modeling techniques are followed by *Cayuga* [10] and *Esper* [1] [27]. Attributes like the selection policies or contiguity options may differ from one system to another. For example, the *SASE+* implementation expands on the principles of *SASE*, allowing the use of Kleene closure as an operator onto a pattern [19], expanding the expressiveness of its language as a result. Moving on more recent implementations, two of the most prominent systems are FlinkCEP [41]- on which we expand further on the rest of this study- and Siddhi [45]. These systems differ in the way they approach modeling, with the former using a sequence of events each one assigned one or more conditions on its attributes and the latter following a method closer to *SASE*, as conditions are stated at the end of the pattern, regardless of the event they refer to.

2.1.2 Logic-based Complex Event Recognition systems

Logic has been used as the basis for systems that detect patterns on streams of real data in different occasions [5] [44] [9] [26]. The process of designing Complex Events based on logic can be divided on two separate types: *Chronicle recognition* and *Event Calculus*.

Chronicle Recognition Chronicle Recognition Systems interpret time relations between SDEs as Complex Events [21]. During recognition, these systems use windows of increasing size to find a successful match with the new events included on each step, based on the chronicle modeled initially [20]. The CER process tends to be computationally complex, as all possible partial matches are being identified by the system. Applications of chronicle recognition include monitoring gas turbines, traffic or even telecommunication networks [32]. Moreover, based on the components of the chronicle matched at any moment, it can generate possible developments, making some sort of prediction on events that have yet to occur. Examples of CER engines following this approach include *TESLA* [16] and *Amit* [2], with the former offering operator for negation and the latter allowing input events to have a duration and not be strictly instantaneous.

Event Calculus As proposed by [14] and [13], Event Calculus can be represented as a methodology for creating and handling automated workflows; the process of CER can easily be regarded as such a workflow. While automata focus on sequences of happenings within the stream, Event Calculus deals with determining the consequences of actions (SDEs) [38]. As any representation of logic, Event Calculus faces the question of the frame problem [36], on the need of providing enough axioms to determine a viable description of the environment for a machine. The essence of this problem is the simple representations of the effects of some actions omitting at the same time the need for further description on their non-existent effects [31] [43]. The solution provided in Event Calculus is concept of inertia, which plays a major role in its understanding. Inertia dictates that a fluent (property that holds values over time) has got a value in a precise point in time, if at a previous point it was assigned this value, triggered by the occurrence of an event, and if no other event has resulted in the change of this value in the meantime [35]. Moreover, the use of negation is also a prominent concept within logic based calculus and thus naturally appears on event calculus.

The RTEC system [6] can be considered an Event Calculus efficient dialect, implementing its fundamental properties. Focusing on fluents, the RTEC engine uses windows and interval manipulation in order to return fluent-value pairs, according to the rules it was provided, in order to detect the maximal intervals where they holds a certain value.

2.2 Comparisons

Regardless of the fact that CER systems are often based on established fields such as automata theory [15] [34] and logic, a strict definition on the semantics of their operators is rarely provided. Therefore, comparing these types of CER engines requires a more elaborate examination of each system's capabilities. Our study focuses on comparing state-of-the-art systems that belong to the two most dominant categories (automata-based and logic-based) CER.

There has been a few studies that include the comparison of several CER systems. A major work devoted in presenting the different aspects of CER, as well as providing a description of several Information Flow Processing (IFP) engines (as they are called in the paper) was presented in [17]. Similarly, surveys such as [18] and [28] present different Stream Processing Engines, comparing the features of such engines, with the latter focusing also on the parallelism capabilities of each system. A different study that

focused mainly on probabilistic engines is [4]. While these papers present a range of systems, our study focuses on two specific implementations, and provides elaborate examples and a thorough examination of different scenarios including potential patterns and streams that may appear during a recognition process.

As the previous approaches to comparing CER systems do not include scenarios of execution of the recognition process, a paper that has more common ground and scope to our own is [23]. This study describes how Event Calculus can be used to calculate and detect intervals where fluents hold a certain value, but also expanding on [12] provides a modeling technique for reasoning based on timed automata, in other words finite automata enhanced with time constraints upon the transitions between states. Moreover, an empirical comparison has been conducted using such a machine, resulting in positive results on such an interpretation of automata. While this work focuses on timed automata, we are examining a more wide category for pattern modeling, provided by the FlinkCEP implementation, that allows these transitions to include all attributes of involved events or even more complex conditions.

Finally, in [3] we find a study that compares two individual systems in both terms of expressiveness and by using an empirical evaluation; with the systems described and examined being RTEC and the SQL-based Esper. After presenting major concepts of both systems, such as inertia and pattern hierarchy, the conclusions of this study denote that implementing and representing patterns from one language to the other, though not a minor task, is possible. Similarly, we attempt to compare two systems by examining whether Complex Events expressed on one system can be translated into the dialect of the other, as well as evaluating their performance over streams of real data.

3. BACKGROUND

In this chapter, we provide a deeper look on the two systems we use in our comparison. First, we present the basics for CER based on automata; listing the capabilities of FlinkCEP and giving some simple examples to display its use. Afterwards, we present the basics regarding Event Calculus and describe how the RTEC system approaches both modeling Complex Events and the recognition process.

3.1 The FlinkCEP System

A state-of-the-art engine, FlinkCEP is provided by the Apache Software Foundation. Because it is built on top of Flink, Apache's own streaming environment, it enjoys all its streaming attributes and can be combined with other streaming implementations, like Kafka's environment. As it is with Flink, the FlinkCEP library is available for both Java and Scala applications ¹.

Flink Streams Flink streams consist of events defined by the user. These events may include several attributes or values assigned to them, as they are implemented and behave as Scala Objects. The stream should be comprised by the same type of Objects; all attributes and methods/functions are available at any point during its process. Furthermore, the Flink environment provides an operator that reorders our current stream based on the values of one of the event attributes. Usually, because CER includes tasks over temporal attributes, we are assigning the time value of each event onto an Object attribute. This way we are able to reorder the stream accordingly if necessary and impose temporal condition within our patterns.

Flink is also able to split our stream into different sub-streams based on the attributes of the input events, as they arrive at the system. Using this feature, the parallelism capabilities of our machine can be used to the fullest, as each thread would undertake to handle certain partitions of the original stream. Here, we should note that the Flink system provides us with the choice of having several operators (components of our streaming application) to be applied with a different parallelism factor than the rest of the system. More precisely, this feature can be used when reading data from a consumer, so that we would avoid an unsorted input stream. Expanding on that last option, we should mention that it can also be used as an effective method for our patterns to be simpler, omitting conditions that ensure all events of the match concern the same entity.

FlinkCEP Patterns All patterns created using FlinkCEP's dialect can be characterized as sequences of events, that may be found within the stream. The way a pattern is structure plays an important role on its semantics. More precisely, the order in which the events are included in our pattern is of decisive importance. Each single event (or more precisely component of our pattern) is characterized and thus can be accessed and retrieved if necessary by a unique name it is assigned during the patterns definition. A simple pattern example, assuming we have a stream describing the progress of the

¹In our study and experiments we use the Scala syntax and environment for FlinkCEP.

velocity of a vehicle, that detects a remarkable change of velocity of this vehicle can be expressed as follows:

```
val changePattern = Pattern.begin("start")
    .followedBy("chng")
    .where((ev, prev) =>
        prev.getEventsForPattern("start").last.getVelocity()
        - ev.getVelocity() > 30)
    .within(Time.seconds(15))
```

In this example we get introduced on multiple capabilities of the FlinkCEP system, including the condition operator, the relaxed contiguity option as well as more complex features. We would be expanding on these system features on the remaining part of this chapter.

Conditions When facing a new event of the stream, the CEP operator should be able to decide whether it should accept it and include it as part of our (partial) match or discard it and move on. The rules that determine the outcome of this decision are included within the conditions that accompany the event on the pattern's definition. These conditions can be divided into two categories: those who include solely the attributes of the event in question and those that involve attributes of multiple events on our match, and use the properties of previously accepted events to produce an answer. The latter set of conditions is defined as *Iterative Conditions*, while the former is called *Simple Conditions*. These conditions are formed by the use of boolean operators between attributes of the including events or even involving the values of external to the CER process variables. Although there is a certain freedom regarding whats included in the conditions, the use of variables in order to keep certain values between the parts of our matches is highly discouraged. The reason behind this last remark, relies on the fact that the values kept would change according to the method the system generates candidate matches. As one might easily deduce, restricting conditions on the attributes of a single event would limit the system significantly in terms of expressiveness. Thankfully, the FlinkCEP system includes an option to retrieve parts (meaning events) of our current partial match, in order to form more complex and meaningful Complex Events, the `getEventsForPattern` function. Deciding on which component to retrieve based on the component name provided by the user, this function is indicated to have varying computational cost, and thus is recommended to limit its use on all implementations. This feature can be used upon already accepted events of the stream, and thus conditions that include future events are not permitted.

Contiguity Options As already mentioned, a major factor that determines the nature of our pattern is the contiguity between its components; i.e. the way each event of the current match succeeds the previous, within the original stream. This type of relation is defined between consecutive components of our pattern. More precisely, each event accepted (except the initial one) must be consistent on the way it follows the previous one already accepted on the partial match, as determined by the pattern's definition. For example on the previous example regarding velocity changes we were using Relaxed Contiguity between the two events accepted, provided by the `followedBy` method, as we allowed irrelevant events to intervene between them within the stream. The available contiguity options provided by the system are the following:

- **Strict Contiguity:** the events of the pattern must be consecutive within the original stream (indicated by the use of `next`).
- **Relaxed Contiguity:** non-matching events are allowed to appear between the accepting ones (indicated by the use of `followedBy`).
- **Non-deterministic Contiguity:** does not terminate the recognition process when finding a matching event, but also investigates other occurrences as further on the stream (indicated by the use of `followedByAny`). This, of course, causes the resulting patterns to have a higher cost in terms of complexity.

We ought to keep in mind that these strategies are applied based solely on the order the stream is provided to our CER engine and not the actual timestamps each event occurred in. A interesting case that we will come across on our further study is the occurrence of simultaneous events within our stream. By definition these events would be given in a particular order within the stream. This order often would carry no real context, but would dramatically alternate the patterns and their structure. Multiple solutions can be practiced to overcome this particularity; several of which we are presenting in the following chapter.

Finally, after determining which attribute of the stream events would be the one that indicates the temporal traits of the input, an additional type of condition is available for our patterns: the `within` method. This method restricts our patterns so that the maximum temporal distance between all events of our match is not be greater than a certain value (measured in seconds). This was displayed on our previous velocity example, as the change should take place within the range of 15 seconds.

Event quantifiers As all CER systems deal with streams that are constantly fed with new information and events, it is realistic to assume that some the occurrence of same typed event consecutively within a pattern would be of useful semantic interpretation. FlinkCEP pattern components are optionally accompanied with an operator that indicates the possible and allowed number of occurrences for each component. For example, going back to our vehicle velocity stream, let us assume that we need to detect the occurrence of 15 consecutive events that indicate a velocity of over 85 mph. This can be easily designed by having 15 separate components to our pattern. The conditions of these all these components would be identical, i.e. the velocity of each one to be greater than 85 mph; so including more than one of these components on the pattern's definition would be without any real meaning. Fortunately, we are able to characterize this component as 'looping' and provide the system with the number we require for it to appear:

```
val fastPattern = Pattern.begin("start")
    .where(ev=>ev.getVelocity() > 85)
    .times(15)
```

Imagine now that we need to detect all situations where this vehicle is exceed 85 mph, and that it does so for more than 15 times but also less than 100 because that would indicate a different phenomenon that our system is not assigned to detect. Furthermore, lets assume that we need to detect a potential immobilization of a vehicle following the above scenario. This Composite Event could be expressed using the optional feature as follows, characterizing our pattern as non-deterministic:

```

val fastStopPattern = Pattern.begin("start")
    .where(ev=>ev.getVelocity()>85)
    .times(15)
    .next("abrupt")
    .where(ev=>ev.getVelocity()==0).optional()

```

After Match Skip Strategies While most systems are including some sort of Consuming Strategy [28], defining the condition under which the machine is done processing an event. The somewhat equivalent strategies defined for this implementation are called the After Match Skip Strategies. These strategies basically indicate to the system the point of the stream where the recognition should resume after having detected a successful match. These strategies are defined in relation to each patterns structure and components. Each strategy applied would have a significant impact on the nature of the pattern and thus the concept it ultimately expresses, because of the fact that some matches could be potentially omitted when applying a more exclusive strategy.

The strategies provided by our system are the following:

- **No Skip**: does not omit a potential match.
- **Skip to next**: does not omit any event and resumes at the very next event after our match, but not attempting to detect another successful match that begins with the same event as the last one.
- **Skip-Past-Last Event**: discards all events that occur within the bounds of the stream that the previous match determines (after its first component and before its last one) and resumes at the very next event. This strategy is proven to be extremely useful when having to deal with Patterns that have a single initiation and a single termination point at the stream, as it can be used to omit any non maximal matches.
- **Skip to first/last (*CompName*)**: the system would resume its CER process at the first/last component of the match assigned the name *CompName*.

Negation in FlinkCEP A common issue when dealing with constant sources of information the concept of negation has been proven to be challenging. While dealing with negation upon the attributes of a single event is quite simple, deciding on similar conditions that involve more than one event is a completely different matter. We will expand on this issue on following chapters, as it has proven to be a major difference between a logic-based and automaton-based systems.

The notion of negation FlinkCEP supports is restricted to the occurrence of events of a particular type within our pattern, not including the bound events of our match. More precisely, we are able to forbid the appearance of a type of event between two consecutive components of our stream (supposing we don't have a strict contiguity). For example, imagine we have to monitor the movement of a vehicle as before, but in this case we want to record where the vehicle is moving after having a velocity greater than 85 mph but

without exceeding it again, until it stop moving. This could be implemented by having the following pattern:

```
val fastDriveStopPattern = Pattern.begin("start")
  .where(ev=>ev.getVelocity()>85)
  .notFollowedBy("not")
  .where(ev=>ev.getVelocity()>85)
  .next("end")
  .where(ev=>ev.getVelocity()==0)
```

As we can see this feature ensures us that no event that follow the “not” format could appear on our match between the “start” and “end” components. Additionally, the FlinkCEP system disallows us of using such a component (assign a notFollowedBy tag) as our final component, because the streams expected as input are of infinite length.

Higher-Level Recognition When dealing with the recognition process of our Flink implementations we should take in consideration the fact that the resulting matches of a CEP operator are actually in the form of a stream. This last fact allows us to apply different types of operators upon the results. Such an option available for our implementations is applying a new CER upon this newly formed results stream. Let us assume that we have the first pattern that detects abrupt accelerations on the vehicle’s speed. A different concept that might be of interest is the detection of consecutive such accelerations that are occurring within 3 hours between them. This scenario can be represented by the following block of stream operators:

```
val changePattern = Pattern.begin[VelocityEvents]("start")
  \ \ type of stream event within brackets
  .followedBy("chng")
  .where((ev, prev)=>
    prev.getEventForPattern("start").last.getVelocity()
    -ev.getVelocity() > 30)
  .within(15)

val CEPResultsStream =
  CEP.pattern(originalStream, changePattern)
//in order to perform the Recognition process
val resultsStream: DataStream[changeMatch] =
  CEPResultsStream.select(...)
  //ommiting function that transforms a match to a new
  Scala Object for simplicity

val metaChangePattern = Pattern.begin[changeMatch]("start")
  .followedBy("end")
  .within(Time.seconds(10800))
```

We can observe that this type of pattern can be characterized as a higher order pattern. Applying consecutive CER processes seems to be more simple than attempting to express such concepts in a single pattern. Moreover, we claim that dealing with Complex Events using a hierarchy of patterns allows us to implement events that would carry great cost if implemented without it. The drawback of this method is the need to handle a different stream, that could potentially be as long as our original. Our empirical studies has shown that a significant amount of our execution process is due to the iteration upon the stream rather than the operators themselves.

3.2 Event Calculus

While examining the fundamentals of the Event Calculus (EC), as introduced by Kowalski and Sergot in 1986 [35], one immediately comes across the concept of fluents. This comes natural as the main purpose of Event Calculus is the study of the effects of events on the values of fluents. *Fluents* are entities that behave like variables over time. A fluent might hold a range of values, one each time, but it is also possible there are moments where it does not have any value.

The behavior of fluents and their values is defined by custom rules. These rules determine which events and under which circumstances could cause an action to be taken upon the value of a certain fluent. They define the initiation and the termination points of a fluent, in other words the moments where a value is assigned or when the fluent is withheld a value. Furthermore, predicates that can be used to decide upon the value of a fluent at a given time are provided. A predicate as such is the $\text{holdsAt}(F=V,T)$, indicating that fluent F holds the value V at time T . The occurrence of an event is expressed using happensAt with similar predicates defined on each different dialect available.

3.2.1 The RTEC System

The RTEC system is based on the same principals as the Event Calculus, hence the results it provides are an attempt to detect maximal intervals where a fluent holds a single value continuously.

Complex Event Modeling The RTEC implementation uses time constrains and interval manipulation to handle the conditions of the patterns and to extract the successful matches. Special predicates have been defined in order to create rules that define Complex Events. These rules are described on Table 1.

Table 1: RTEC predicates and operators used for rules of Complex Events (after [6]).

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

Patterns can be expressed by fluents determined using rules provided by RTEC. The conditions on these rules can refer to either the timepoint where they are triggered, or either facts about other timepoints or entities. These rules are triggered by the occurrence of a event, but are not limited to it, as they can also refer to other events, the negation of an event or even the value of another fluent. As the two types of fluents supported by Event Calculus and RTEC are *Simple Fluents* and *Statically Determined Fluents* (SDFs) there are two corresponding types of rules for such patterns. The first, consist of one or more initiation rules along with one or more termination rules for the pattern. These rules refer to the point where a fluent gets initiated with a certain value and where this value get terminated. The second type of rules are the ones regarding SDFs. As opposed to how definitions of Simple Fluents are comprised by possibly multiple rules, a SDF gets determined by a single rule that includes all events and conditions that need to hold for the fluent to hold a certain value.

Below an example of a Simple Fluent defined in RTEC is provided. This fluent indicates a period where a vehicle is moving with a high velocity, according to the aforementioned rules. It is assigned the true value when the vehicle (*Veh* in our pattern) moves with a greater speed than 75, and gets terminated (does not hold this value) when its speed drops from 45. We are assuming that our input stream is consisted of messages that refer to the vehicle's speed over time.

```
initiatedAt(highSpeed(Veh)=true, T):-
  happensAt(speed(Veh, CurSpeed), T),
  CurSpeed > 75.
```

```

terminatedAt(highSpeed(Veh)=true , T):–
  happensAt(speed(Veh, CurSpeed),T),
  CurSpeed < 45.

```

Recognition A main component of Stream Processing is the use of temporal boundaries, known as windows, upon its operators. Moreover, the use of windows upon a process of a CEP system partitions the stream of data into smaller streams in order for the operator in question to be applied on each one separately. This extenuates the load of the system has to manage in each step of the process. The recognition process of RTEC is capable of such features in order to expedite the recognition process and optimize the system's performance. The windows provided are sliding, meaning that they have a fixed, predefined size, and progressing throughout the timestream. Depending on the needs of the user, the windows can be overlapping, meaning that there would be common intervals between consecutive windows, or not. It is important to note that, although the dataset is broken into separate piece of data, the RTEC system is implemented so that the use of windows does not alter the expressive power of our patterns and the recognition process; meaning that the optional use of windows and their potential size does not affect the final results of the recognition, besides the execution time performance.

The RTEC recognition process consists of the following steps for each window. At first, the bounds of the window get determined. Afterwards, all events that belong to a previous window, i.e. their timestamp is lower than the left bound of our current window, get deleted (retracted) and those that come with the current get asserted. Moving forward, it computes all valid holdsFor and holdsAt predicates for all patterns defined. Lastly, the maximal intervals where fluents hold a certain value are calculated and returned.

4. THEORETICAL COMPARISON

RTEC patterns are expressed by two types of fluents: these that are defined by initiation and termination rules, called Simple Fluents, and Statically Determined Fluents (SDF), determined by conditions that need to hold throughout the matching interval. Simple Fluents may be Boolean or multi-valued; in this study we focus on the former. In this chapter we examine whether FlinkCEP is able to generate equivalent patterns, as well as listing the difficulties of this process. In each section, we describe an indicative RTEC pattern, an attempt to simulate its behaviour using FlinkCEP and the challenges we come across in each case.

4.1 Unbounded Intervals

Simple Fluent matches are defined by an initiation point on the stream; the fluent holds a certain value until it gets terminated. In case no termination rule gets satisfied, RTEC returns an interval that does not include a termination point.

Imagine we have a simple pattern that is defined by the occurrence of two events, one at initiation and another at termination.

RTEC

```
initiatedAt (mypatt(X)=true ,T):-
    happensAt(a(X),T) .
terminatedAt (mypatt(X)=true ,T):-
    happensAt(b(X),T) .
```

FlinkCEP

```
val mypatt = Pattern.begin("start").where(ev=>ev.get()=='a')
    .followedBy("end").where(ev=>ev.get()=='b')
```

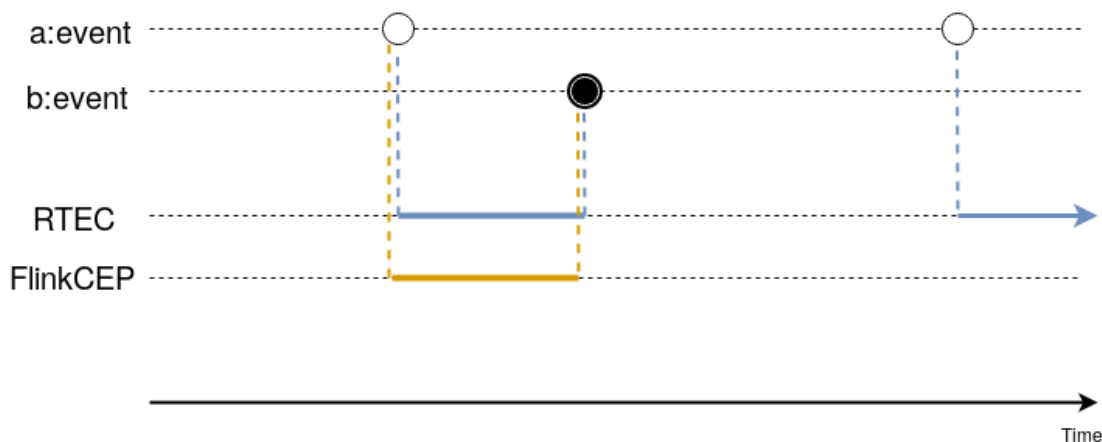


Figure 1: RTEC is able to detect unbounded matches, in case the stream does not include an ending point, while FlinkCEP cannot simulate this behavior.

In order to simulate this behavior using FlinkCEP we would have to mark the ending part of the pattern as optional, for the system to be able to stop and return a match. Unfortunately,

using this technique the FlinkCEP also includes an unbounded match even if the ending point occurs, thus including incorrect results.

A different approach would be to create two distinct patterns: one that only detects the full (that include a termination) matches and another that would return all potential unbounded intervals. This solution requires an additional step that would discard all unbounded intervals that also correspond to a *full* match.

4.2 Simultaneous initiation and termination

Often streams include events that initiate a pattern simultaneously with events that terminate it. This may be caused by the nature of the stream or due to noise in the dataset. During RTEC's recognition process matches that are comprised of simultaneous initiation and termination points are discarded as the system does not detect a single timepoint where the fluent in question holds a value. Suppose we have the same RTEC pattern as in the previous section.

RTEC

```
initiatedAt (mypatt(X)=true ,T):-
    happensAt(a(X),T) .
terminatedAt (mypatt(X)=true ,T):-
    happensAt(b(X),T) .
```

FlinkCEP

```
val mypatt = Pattern.begin("start").where(ev=>ev.get()== 'a')
    .followedBy("end").where((ev, prev)=>{
        ev.get()== 'b' &&
        prev.getEventsForPattern("start").head.getTimeStamp()
            != ev.getTimeStamp() })
```

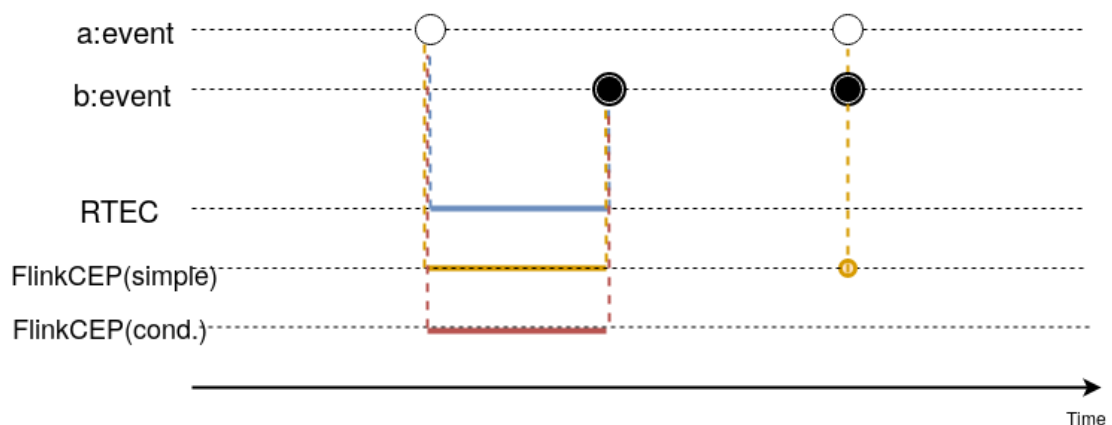


Figure 2: While FlinkCEP would accept a match that is defined by simultaneous initiations and terminations, RTEC automatically filters such matches. In order to avoid these matches we should include an additional temporal condition.

In order to handle such scenarios using FlinkCEP, we can discard all matches that include a single timestamp during post-processing, or we may include an additional condition to

our pattern. This condition would demand the timestamps for the parts of our match to be different, through the `getEventsForPattern` function. It is noted on FlinkCEP's documentation that the use of this function should be limited as its computational cost varies. Furthermore, in case the termination event appears first within the input stream (followed by the simultaneous initiation event), the aforementioned FlinkCEP approach would not discard the match and create a match with the next termination point appearing. In conclusion, handling instantaneous matches is a challenging tasks that requires additional computations while using FlinkCEP .

4.3 Simultaneous Events

On several occasions, patterns include events that occur at the same timepoint. These patterns appear usually when dealing with relational Complex Events between multiple entities. The RTEC system deals with simultaneous events regardless of the order they appear within the stream, as it focuses on the occurrence of the events and not their sequencing. On the other hand, the order in which these events appear plays a major role during the design of the patterns as well as the recognition process of FlinkCEP . For example, in Figure 3, both streams contain the same information but are not treated as equivalent by the FlinkCEP system. This occurs because the first two (simultaneous) events of both streams do not appear with the same order. This remark comes naturally as FlinkCEP evaluation process is based on automata.

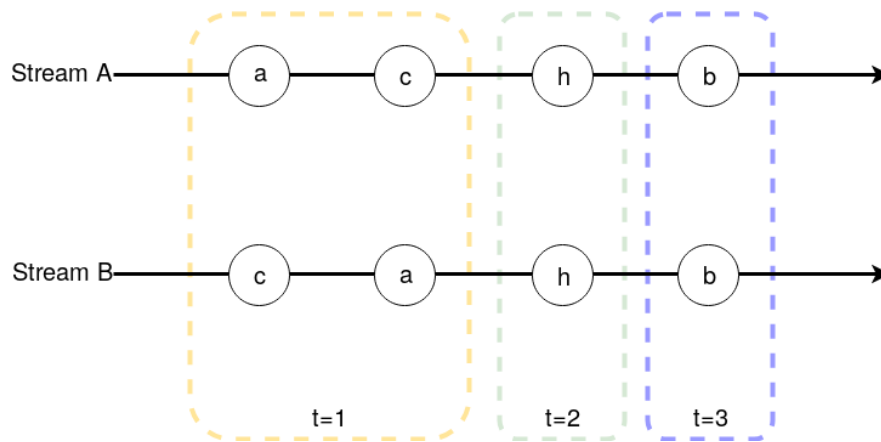


Figure 3: The two streams include the same information but are treated differently by FlinkCEP . A pattern that requires for the (a) and (c) events to occur simultaneously requires a more complex approach for FlinkCEP .

The importance of the order between simultaneous events can be demonstrated by the following scenario: imagine a pattern that requires multiple events to happen at its initiation. Forming a simple sequence of these events while creating the FlinkCEP equivalent Complex Event would not be a sufficient approach, as it is implied that the order between them would be as declared within this pattern in all cases.

Suppose we have the following RTEC pattern, that also includes negation on the occurrence of events.

RTEC

```
initiatedAt (mypatt(X)=true ,T):—
```

```

happensAt(a(X),T),
\+ happensAt(m(X),T).
terminatedAt(mypatt(X)=true,T):–
happensAt(b(X),T),
\+ happensAt(n(X),T).

```

Handling simultaneous events on FlinkCEP can be achieved by the use of bookmark events, in order to manage all events that occur on a certain timestamp. We present a pattern that uses this method.

FlinkCEP

```

val mypatt = Pattern.begin("book1")
    .notFollowedBy("not1").where(x=>x.get()=="m")
    .followedBy("start").where((key, ev)=>{
        x.get()=="a" &&
        ev.getEventsForPattern("book1").last.getTimestamp()
        < key.getTimestamp()
    })
    .next("middles").oneOrMore.optional
    .next("end").where((key, ev)=>{
        key.get()=="b" && ev.getEventsForPattern("middles")
            .count(x=> x.get()=="m"
                && x.getTimestamp()==key.getTimestamp())==0

        && ev.getEventsForPattern("middles")
            .count(x=> x.get()=="n" && x.getTimestamp()==
                ev.getEventsForPattern("start").last.getTimestamp())==0

        && ev.getEventsForPattern("start").last.getTimestamp()
            < key.getTimestamp()
    })
    .notFollowedBy("not2").where(x=>x.get()=="n")
    .followedBy("book2").where((key, ev)=>
        ev.getEventsForPattern("end").last.getTimestamp()
        < key.getTimestamp())

```

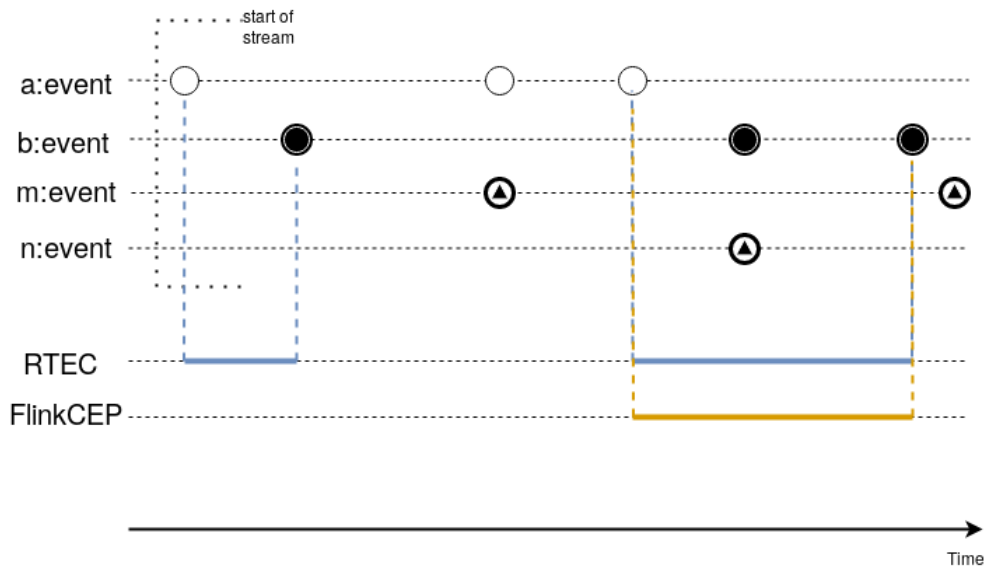


Figure 4: The two streams include the same information but are treated differently by FlinkCEP . A pattern that requires for the (a) and (c) events to occur simultaneously requires a more complex approach for FlinkCEP . The FlinkCEP implementation cannot detect the first match as no events are included into the stream in order to be used as bookmarks.

This approach for the FlinkCEP pattern suffers from being too complex compared to its RTEC equivalent; also using the `getEventsForPattern` method extensively throughout the recognition. Furthermore, the dependence on other events that are practically irrelevant to our match, may cause significant delays during real-time recognition when dealing with sparse streams.

MegaEvents An alternative approach is the use of ‘MegaEvents’. These new types of events are comprised of events that occur simultaneously within the original stream. By definition these ‘MegaEvents’, include all information of the events they are composed of, they are defined unequivocally by these events and include their common timestamp. An implementation of this concept would be to create a ‘TupleEvent’ data structure: pairs of all possible simultaneous event combinations. These events would be created in a preprocessing stage and result in a new stream upon which the CER process would be applied. For example, the first two events on both stream of Figure 3 will be translated as a tuple event: $TupleEvent([a, c], 1)$. On the other hand, the inclusion of this concept creates some issues during recognition, as it can result in a stream with length much greater than the original; affecting the execution time performance of the FlinkCEP system. Moreover, the inclusion of multiple events as single components of our match may require additional conditions in order to determine which of them are useful during recognition.

Ordering simultaneous events In order to avoid the issues arising when dealing with simultaneous events, we contemplate on solutions that presuppose an order between them. This order should be known before the developing of the pattern and often is worth the preprocessing step. Unfortunately, imposing such orders on our streams may not be possible because of conflicts between rules of our patterns. We will be using the following example, to demonstrate the issues that may occur when we impose ordering rule based on pattern attributes.

When including conditions that require the absence of an event, the order of simultaneous events plays a major role during the designing of our pattern. A possible order of simultaneous events that would benefit this process is the following: all events that appear without a negation operator attached to them (within the rules of our pattern) are being ordered first, followed by the rest. When dealing with the following pattern we come to an impasse, as event (a) appears on both types of conditions. In this case, when a pair of **a-b** occurs simultaneously, we wouldn't be able to determine a proper ordering.

RTEC

```
initiatedAt (mypatt(X)=true ,T):-
    happensAt(a(X),T) ,
    \+ happensAt(b(X),T) .
terminatedAt (mypatt(X)=true ,T):-
    happensAt(b(X),T) ,
    \+ happensAt(a(X),T) .
```

In conclusion, in order to deal with simultaneous events one may create patterns that use bookmarks, risking delays during recognition upon sparse streams. Furthermore, another solution includes transforming the input stream of events onto a different that includes entities more complex and comprehensive; requiring a preprocessing step and possibly additional computations during evaluation. Lastly, the imposition of an order for the simultaneous events may serve us in several occasions, as long as this order is pattern-independent.

4.4 Relations between patterns

Certain patterns depend on the values other fluents hold during their initiation or termination; creating a hierarchy between them. This property can be expressed by the use of the `holdsAt` option in RTEC . Trying to translate this attribute on the FlinkCEP system, has proven to be no simple task. In order to detect these patterns, RTEC first computes the intervals where the prerequisite patterns hold a value, and then moves on the pattern that depends on them. Simulating the RTEC 's method would require for FlinkCEP to iterate over the stream multiple times, as well as merge streams that include both input events and complex events matches, as in all cases the conditions of our patterns involve both types of input.

An alternative to simulating RTEC 's behavior would be to integrate the components of all involved RTEC patterns into a single FlinkCEP equivalent. Besides the fact that these FlinkCEP translations tend to be too complicated, the following example shows that there exist scenarios where this approach is not feasible. Imagine we have a pattern that only gets initiated when an (a) events occurs and when a different pattern does not hold. Two FlinkCEP patterns are provided.

RTEC

```
initiatedAt (preq(X)=true ,T):-
    happensAt(s(X),T) .
terminatedAt (preq(X)=true ,T):-
    happensAt(t(X),T) .
....
```

```

....
initiatedAt (mypatt(X)=true ,T):–
    happensAt(a(X),T) ,
    \+ holdsAt (preq(X)=true ,T) .
terminatedAt (mypatt(X)=true ,T):–
    happensAt(b(X),T) .

```

FlinkCEP [1]

```

val mypatt1 = Pattern.begin[MyEvent]("preq")
    .where(x=>x.getAnnot()=="t")
    .notFollowedBy("not").where((key, ev)=>{
        key.getAnnot()=="s" &&
        key.getTimestamp() >
            ev.getEventsForPattern("preq").last.getTimestamp()
    })
    .followedBy("start").where((key, ev)=>{
        key.getAnnot()=="a" &&
        key.getTimestamp() >
            ev.getEventsForPattern("preq").last.getTimestamp()
    })
    .followedBy("end").where((key, ev)=>{
        key.getAnnot()=="b" &&
        key.getTimestamp() >
            ev.getEventsForPattern("start").last.getTimestamp()
    })

```

FlinkCEP [2]

```

val alex = Pattern.begin[MyEvent]("preq")
    .where(x=>x.getAnnot()=="s").optional
    .next("start").where((ev, prev)=>{
        ev.getAnnot()=="a" &&
        prev.getEventsForPattern("preq").isEmpty
    })
    .followedBy("end").where((key, ev)=>
        key.getAnnot()=="b" &&
        key.getTimestamp() >
            ev.getEventsForPattern("start").last.getTimestamp()
    })

```

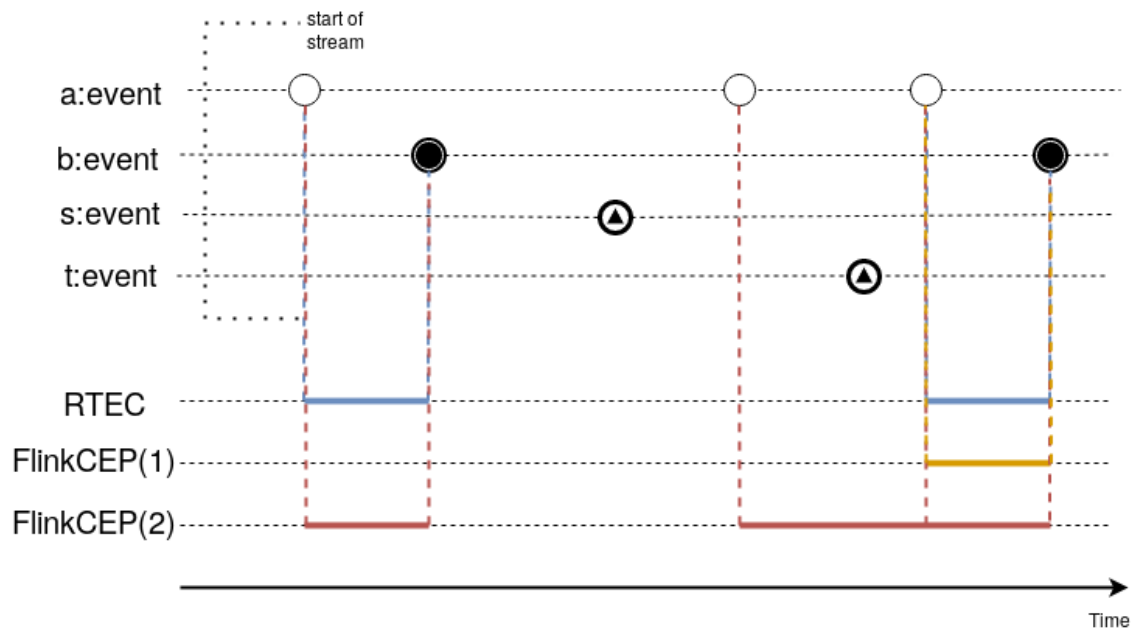


Figure 5: The prerequisite pattern (defined by (s) and (t)) should not hold during initiation of the pattern we are trying to detect. The first FlinkCEP approach would be to include the termination of the prerequisite as part of the pattern, missing the first match. The second, marks the first pattern's occurrence as optional allowing for the system to ignore it even if it appears.

The reason this pattern cannot be translated properly, is the fact that FlinkCEP allows access solely to events that are part of our current match during recognition. In logic programming the *Negation as failure* rule is used, meaning that the negation operator succeeds on the absence of the event in question. Simulating this using FlinkCEP is practically impossible as we would need to store the whole stream until the point of the query.

4.5 Conclusions

In this section we summarize of all differences between the two CER engines. These differences refer to the expressiveness of both systems and should be indicative on which one a user should prefer, depending on the nature of patterns that need to be implemented.

- **Query capabilities** The main difference observed concerns the capabilities of each system to consult on other events and facts during the recognition process. More precisely, the RTEC system is able to perform several queries regarding past events of the stream and examine their attributes at any point within the pattern. On the contrary, FlinkCEP is only able to access events that are part of its current (partial) match of the pattern during recognition. This fact leads to much more complex patterns when using the latter system. Moreover, the space complexity of FlinkCEP patterns that access past events may end up high when dealing with dense streams. Additionally, handling simultaneous events within our patterns creates the need of altering the input stream for FlinkCEP , by using techniques such as imposing an order onto these simultaneous events.
- **Unbounded results** A further difference observed, deals with the fact that the RTEC system is capable of accepting matches that do not include a termination point on our stream, and thus result in unbounded intervals. Such matches can not be efficiently simulated using FlinkCEP , as any such attempt would include matches that falsely ignore events of the input stream.
- **Negation** Moreover, the inclusion of negation within conditions of our Complex Event, creates several incompatibilities between the two systems. RTEC 's interpretation of the negation upon the an event or a fluent can not be translated to FlinkCEP 's dialect. The reason behind this originates from the inability to access previous data without storing a significant portion of the stream during the recognition process, which is a computationally unacceptable solution.
- **Higher-order recognition and pattern hierarchies** A remarkable aspect of the RTEC system is its ability to provide the template for patterns that involve events of the original stream along with conditions on the values of other Complex Events. The FlinkCEP system, on the other hand, separates the recognition of other patterns from the original stream and hence our current pattern. Moreover, even if Complex Events that refer solely other patterns can be designed easily, combining their results with the original stream is a quite difficult and time consuming task.

5. EMPIRICAL COMPARISON

In this chapter, after the examination of possible scenarios where the two systems may differ, we evaluate their importance by performing several experiments upon real life streaming applications. We present the datasets used, the concepts implemented as Complex Events and evaluate the results of these implementations. In total we would be focusing on two datasets where the RTEC system has already been used. These are: the Brest dataset [40] that concerns the monitoring of vessels and the CAVIAR dataset, that provides different scenarios of interactions between individuals using Surveillance Recordings.

5.1 Experimental Setup

In this section we describe the process of our experiments and subsequent comparisons. First, we present the datasets, the types of events within and their temporal attributes. Afterwards, we depict some of the RTEC Complex Events considered for the comparison along with their FlinkCEP equivalents. Lastly, we provide the aspects on which the results were evaluated.

5.1.1 The Datasets

RTEC has been used in numerous projects, including fleet management and maritime monitoring [39]. In this section we present the two datasets we are basing our empirical evaluation on, providing with a few details of the semantics of the SDEs included and the features of each stream.

Maritime Dataset Throughout different types of applications, tracking and monitoring maritime information and vessel movement appears as an important aspect. Most systems that need that type of information use the Automatic Identification System (AIS); a technology for tracking the movement of vessels and locating vessels at sea.

Handling messages transmitted from vessels, a main use and purpose of a recognition system is the avoidance of collisions and ensuring safety across sea traffic. Systems that handle these messages, also use databases including Static data about vessels and Spatial Information. These data include the types of vessels based in its Maritime Mobile Service Identity (MMSI), the speed limits of each vessel type, information about the area the vessels enters or leaves based on coordinates and others.

The AIS messages include information about the vessel that transmits them, like its MMSI, the moment (measured in POSIX / UNIX Epoch time) the message was recorded, along with other data regarding the vessel's velocity, its coordinates etc. The Brest dataset includes information regarding the movement of vessels appearing within the Celtic sea, the Channel and Bay of Biscay in France. It covers in total the time span of six months, beginning at October 1st, 2015 and ending with March 31st, 2016.

In order to have these messages translated and parsed into a form accessible from the RTEC system, a preprocess stage has being applied onto the stream. This process

resulted into some of the messages splitting and producing more than one events on our stream. For example, a velocity message for a vessel may be produced along with another event regarding the vessels movement within an area. At the end of the process, we are provided a stream of different types of events that can be understood by the RTEC system, but also parsed into a Flink stream.

Each one of these events carries different kind of information; for example as the velocity event gives us the speed of the Vessel (measured in knots) and the *coord* event gives us its coordinates. The *change_speed* events are provided when a vessel changes its velocity and the *slow_motion* one when a vessel begins a low speed movement, accordingly. Also, when a ship stops (and when it starts moving again) the respective *stop* event is provided; as well as when a vessel changes its heading. Finally, as the Brest map is partitioned into areas, each area is marked and given a name and gets assigned a type. The type of each area may be one of the following: *anchorage*, *fishing*, *natura*, *nearCoast*, *nearCoast5k*, *nearPorts*, indicating the nature of the area. These areas may be overlapping, as some areas lie within others. When a vessel enters or leaves a certain area, the respective (*entersArea*, *leavesArea*) event occurs. These events include the area ID, from which we can deduce the area type. The proximity event gives us spatial information about the relevant positions of vessels that happen to be near, is the only one that includes more than one vessel, and so carries both of their MMSI keys.

Below we present some information regarding the size of our data stream (full 6 months), along with the occurrences for all SDEs types:

Table 2: Temporal, spatial and entity attributes of Brest dataset.

Time Confines (in Epoch format)	
Starting point	1443650401
Ending (last) point	1459461588
Human Time equivalent (on G.M.T.)	
Starting point	Wednesday, September 30, 2015 10:00:01 PM
Ending (last) point	Thursday, March 31, 2016 9:59:50 PM
Data Time Range	15811189 seconds
Types of areas in total	6
Areas in total	1805
Types of vessels in total	37
Vessels in total	5055

Table 3: Table of Simple Derived Events appearing as input for the Maritime dataset, originating from Brest. All input events are instantaneous except 'proximity'. The first three types of events are a result of a spatial preprocessing; the next two originate directly from the AIS messages, while the rest are produced by the trajectory synopsis generator.

SDE Type		Occurrences (#)
Spatial	entersArea	169419
	leavesArea	142575
	proximity	62138
Critical Events	velocity	16263766
	coord	16262944
	change_in_heading	3588015
	change_in_speed_start	777192
	change_in_speed_end	773121
	gap_start	88752
	gap_end	55273
	slow_motion_start	161076
	slow_motion_end	158290
	stop_start	379550
	stop_end	371452
Critical Events in total		6352721
Total number of SDEs		39253563
AIS messages in total		18M (approx.)

Surveillance Dataset The CAVIAR Project (CAVIAR: Context Aware Vision using Image-based Active Recognition)¹ provides a dataset comprised of the representation of interactions between several entities. These scenarios include the meeting of two individuals, a fight between two people, a person leaving an object and others. Each scenario is in a video format, and is also represented as a stream of its frames, including information about the involved entities, as well as having information about the context of each person's/group of persons actions, i.e. the ground truth for a possible recognition process, as noted in [24] and the dataset's documentation². Additional information about the events included within the dataset can be found on the following tables.

Table 4: Entity and SDE attributes of CAVIAR dataset.

Input Stream

Entities(persons) in total	10
Types of SDEs in total	4
Types of SDEs	movement coord orientation appearance
Types of <i>movement</i> SDEs in total	6
Types of <i>appearance</i> SDEs in total	4

Ground Truth

Situation tags in total	7
Types of Situations	walking immobile drop down none fighting browsing meeting
Context tags in total	9
Types of Contexts	joining interacting leaving victim fighting inactive split up moving none browsing

¹<http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/>.

²http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/gt_file_format.txt.

Table 5: Table of Simple Derived Events appearing within the CAVIAR dataset, along with number of Ground Truth events.*Input Stream*

SDE Type	CAVIAR	10×CAVIAR
movement		
active	5358	53580
inactive	9829	98290
walking	29041	290410
running	807	8070
abrupt	590	5900
none	1	10
coord	45626	456260
orientation	45626	456260
appearacne		
appearance	45333	453330
disappearance	1	10
appear	150	1500
disappear	142	1420
Total number of SDEs	182504	1825040
Video frames in total	25154	251540
Tuples in total	29439	294390

Ground Truth

Situation Tag		
Single Entities	CAVIAR	10×CAVIAR
Moving	15143	151430
Inactive	2867	28670
Browsing	679	6790
None	1	10
Group Entities	CAVIAR	10×CAVIAR
Split	226	2260
Fight	630	6300
Join	558	5580
Interact	301	3010
Left_victim	42	420
Move	1313	13130
Context Tag		
Single Entities	CAVIAR	10×CAVIAR
Browsing	1934	19340
Immobile	5598	55980
Walking	9742	97420
Drop_down	1415	14150
None	1	10
Group Entities	CAVIAR	10×CAVIAR
Fight	1086	10860
Meet	1671	16710
None *(Moving)	313	3130

5.1.2 Complex Events and their Implementations

Complex Events for the Maritime Dataset Having numerous available Complex Events defined for previous and current maritime projects [42], below lies a hierarchy between all these patterns. The Complex Events have been implemented and designed as RTEC fluents and thus our goal was to translate them into their FlinkCEP equivalent.

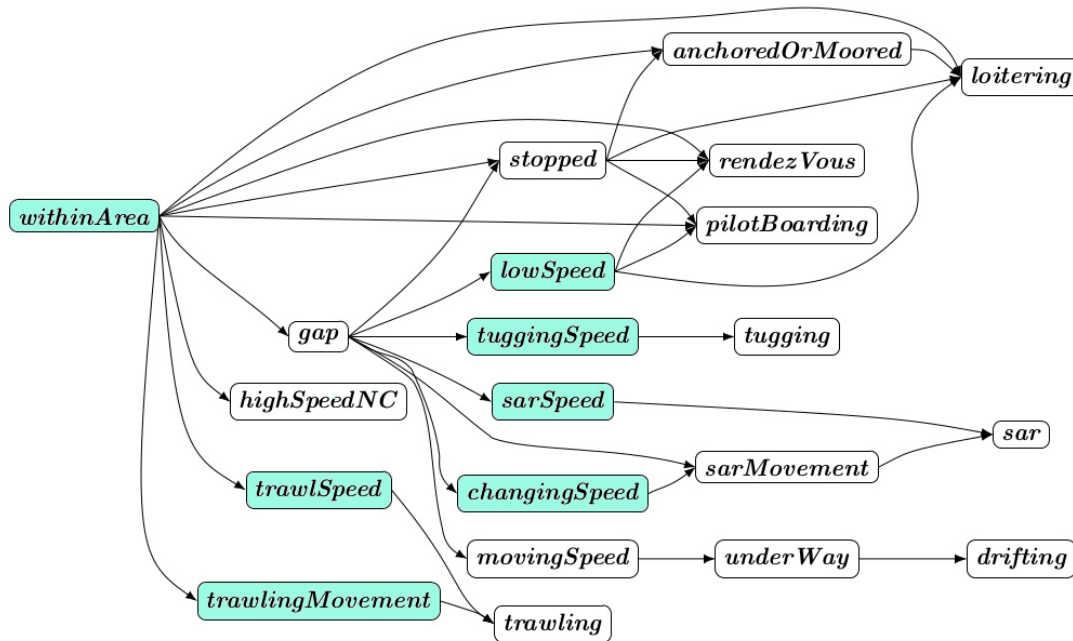


Figure 6: The maritime patterns hierarchy, (after [39]). The patterns used for our evaluation are highlighted appropriately.

All Complex Events examined in this dataset involve only a single vessel. In order to avoid conditions that require events to refer to the same vessel function on our FlinkCEP we decided to partition the stream based on the MMSI tag (the id of each vessel) of all events, for our FlinkCEP implementations. Below we present a few indicative examples for translating these RTEC patterns. First we present a fairly simple pattern (*withinArea*); then we also provide the implementations for the more complex *tuggingSpeed* pattern. All patterns used from this dataset refer to a single vessel.

- **Within Area:**

RTEC

```

initiatedAt(withinArea(Vessel, AreaType)=true, T) :-
    happensAt(entersArea(Vessel, Area), T),
    areaType(Area, AreaType).
  
```

```

terminatedAt(withinArea(Vessel, AreaType)=true, T) :-
    happensAt(leavesArea(Vessel, Area), T),
    areaType(Area, AreaType).
  
```

```
terminatedAt(withinArea(Vessel, _AreaType)=true, T) :-
    happensAt(gap_start(Vessel), T).
```

FlinkCEP

```
val withinAreaKeyed =
    Pattern.begin[MyEvent]("start", skipPastLast)
        .where(ev=>ev.getAnnot()=="entersArea")
        .followedBy("end").where((key, ev)=>{
            val matchStart =
                ev.getEventsForPattern("start").head
                (key.getAnnot()=="gap_start") ||
                ((key.getAnnot()=="leavesArea") &&
                 (key.getAreaType()==matchStart.getAreaType()))
            })
```

• Tugging Speed (single Vessel):

RTEC

```
initiatedAt(gap(Vessel)=nearPorts, T) :-
    happensAt(gap_start(Vessel), T),
    holdsAt(withinArea(Vessel, nearPorts)=true, T).

initiatedAt(gap(Vessel)=farFromPorts, T) :-
    happensAt(gap_start(Vessel), T),
    \+ holdsAt(withinArea(Vessel, nearPorts)=true, T).

terminatedAt(gap(Vessel)=_PortStatus, T) :-
    happensAt(gap_end(Vessel), T).

initiatedAt(tuggingSpeed(Vessel)=true, T) :-
    happensAt(velocity(Vessel, Speed, _, _), T),
    thresholds(tuggingMin, TuggingMin),
    thresholds(tuggingMax, TuggingMax),
    inRange(Speed, TuggingMin, TuggingMax).

terminatedAt(tuggingSpeed(Vessel)=true, T) :-
    happensAt(velocity(Vessel, Speed, _, _), T),
    thresholds(tuggingMin, TuggingMin),
    thresholds(tuggingMax, TuggingMax),
    \+ inRange(Speed, TuggingMin, TuggingMax).

terminatedAt(tuggingSpeed(Vessel)=true, T) :-
    happensAt(start(gap(Vessel)=_Status), T).
```

FlinkCEP

```
val tuggingMin = 1.2
val tuggingMax = 15.0
val tuggingSpeedBKeyed=
    Pattern.begin[MyEvent]("start", skipPastLast)
        .where(ev=>{
```

```

    val speed = ev.getSpeed()
    (speed!=speedInit) && (speed<tuggingMax) &&
    (speed>tuggingMin)
  })
  .followedBy("end").where(ev=>{
    val speed = ev.getSpeed()
    (ev.getAnnot()=="gap_start") || ((speed!=speedInit) &&
    ((speed>tuggingMax) || (speed<tuggingMin)))
  })

```

Complex Events for the Surveillance Dataset While the patterns that involve only one person/entity can be addressed in a direct way, the patterns that express a relational activity between multiple entities need a different approach. More specifically, these patterns need to detect the occurrence of some SDEs about multiple persons, and happening simultaneously in most cases. This last fact implies that events of the stream would come in an order but the timestamps of continuous SDEs would often be the same; as a result when detecting these kind of patterns there is a need to eliminate the sense of order between simultaneous events. In order to achieve this we include the concept of **MegaEvents** in our FlinkCEP implementation. While several types of MegaEvents can be implemented, we propose **TupleEvents**, which are basically tuples of all possible combinations of SDEs that occur simultaneously. The nature of these TupleEvents is defined by a factor, that determines the number of SDEs included on each tuple created. In experiments we chose a factor of two (2). For example, supposing we have the following stream (timestamps indicated in parentheses) is provided:

Event[A](1), Event[A](2), Event[B](2), Event[C](2), Event[A](3), Event[B](3)

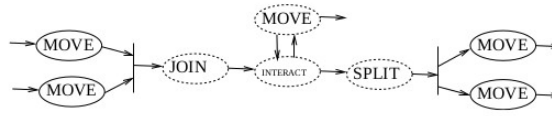
having a TupleFactor of '2' events per MegaEvent, we would create the following MegaEvents(ME) stream:

MegaEvent[AB](2), MegaEvent[AC](2), MegaEvent[BC](2), MegaEvent[AB](3)

Along with the implementation of the RTEC system, several pattern implementations concerning the CAVIAR project have been developed³. We focused on the Simple Fluent based pattern of meeting between two people. The corresponding FlinkCEP implementations of these concepts were developed for the purpose of our comparison; they are listed below.

A concatenated version of all videos have been used for creating an input stream of frames and events. The resulting stream is consisted of approximately **45,000** entity events. On the other hand, the Tuples stream used by FlinkCEP contains about **29.000** of such events. For the purpose of a more thorough comparison of both systems capabilities we also created a second dataset, emanating from the originally provided one. This new dataset is simply a repetition of the former in sequence. We decided on using the initial stream **ten(10)** times and thus creating a dataset of **450,000** events, with the corresponding TuplesStream being close to **295.000** SDEs. We are referring to these two streams as CAVIAR for the simple sequence of the videos provided and as $10 \times \text{CAVIAR}$ for our own extended version.

³<https://github.com/aartikis/RTEC>.

Two people meet:

MOVE (individual): Walker/Walking
 MOVE (group): Walkers/Movement
 JOIN: Meeters/Movement
 INTERACT: Meeters/{Active,Inactive}
 SPLIT: Meeters/Movement

Figure 7: Two people meet / Meeting Context, (after CAVIAR's documentation ⁴).

RTEC

```

/* *****
 *      CLOSE      *
 * ***** */

holdsFor(close(Id1,Id2,24)=true, I) :-
  holdsFor(distance(Id1,Id2,24)=true, I).

holdsFor(close(Id1,Id2,25)=true, I) :-
  holdsFor(close(Id1,Id2,24)=true, I1),
  holdsFor(distance(Id1,Id2,25)=true, I2),
  union_all([I1,I2], I).

holdsFor(close(Id1,Id2,30)=true, I) :-
  holdsFor(close(Id1,Id2,25)=true, I1),
  holdsFor(distance(Id1,Id2,30)=true, I2),
  union_all([I1,I2], I).

holdsFor(close(Id1,Id2,34)=true, I) :-
  holdsFor(close(Id1,Id2,30)=true, I1),
  holdsFor(distance(Id1,Id2,34)=true, I2),
  union_all([I1,I2], I).

holdsFor(close(Id1,Id2,Threshold)=false, I) :-
  holdsFor(close(Id1,Id2,Threshold)=true, I1),
  complement_all([I1], I).

/* *****
 *      MEETING    *
 * ***** */

```

⁴<http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/labelingstates.pdf>

```

% ——— initiate meeting

initiatedAt(meeting(P1,P2)=true , T) :-
  happensAt(start(greeting1(P1,P2)=true) , T) ,
  \+ happensAt(disappear(P1) , T) ,
  \+ happensAt(disappear(P2) , T) .

% greeting1

holdsFor(greeting1(P1,P2)=true , I) :-
  holdsFor(activeOrInactivePerson(P1)=true , IA1) ,
  % optional optimization check
  \+ IA1=[],
  holdsFor(activeOrInactivePerson(P2)=true , IA2) ,
  % optional optimization check
  \+ IA2=[],
  holdsFor(close(P1,P2,25)=true , IC) ,
  % optional optimisation check
  \+ IC=[],
  intersect_all([IA1 , IA2 , IC] , I) ,
  \+ I=[],
  ! .

% the rule below is the result of the above optimisation checks
holdsFor(greeting1(_P1,_P2)=true , []) .

% activeOrInactivePersion

holdsFor(activeOrInactivePerson(P)=true , I) :-
  holdsFor(active(P)=true , IA) ,
  holdsFor(inactive(P)=true , In) ,
  holdsFor(walking(P)=true , IW) ,
  union_all([IA , In , IW] , I) .

% ——— terminate meeting

% run

initiatedAt(meeting(P1,_P2)=false , T) :-
  happensAt(start(running(P1)=true) , T) .

initiatedAt(meeting(_P1,P2)=false , T) :-
  happensAt(start(running(P2)=true) , T) .

% move abruptly

initiatedAt(meeting(P1,_P2)=false , T) :-
  happensAt(start(abrupt(P1)=true) , T) .

initiatedAt(meeting(_P1,P2)=false , T) :-

```

```

happensAt( start( abrupt(P2)=true ) , T) .

initiatedAt( meeting( P1,_P2)=false , T) :-
  happensAt( disappear(P1) , T) .

initiatedAt( meeting( _P1,P2)=false , T) :-
  happensAt( disappear(P2) , T) .

% move away from each other
initiatedAt( meeting( P1,P2)=false , T) :-
  happensAt( start( close( P1,P2,34)=false ) , T) .

```

FlinkCEP

```

val meetingKeyed =
Pattern.begin[TupleMegaEvent]( "start" , skipPastLast)
  .where( ev=>ev.getDistance()<=25 &&
    ev.notExistsApp( "disappear" ) &&
    ev.sdes.count( x=>x.getAnnot()=="active" ||
    x.getAnnot()=="inactive" || x.getAnnot()=="walking" )==2 )
  .followedBy( "end" ).where( ev=> ev.getDistance()>34 ||
    ev.happens( "abrupt" ) || ev.happens( "running" ) ||
    ev.existsApp( "disappear" ) )

```

While comparing the patterns for several concepts, we noticed that in all cases the FlinkCEP equivalent is significantly shorter than the one for RTEC . A main reason for that fact is the complexity of creating sequence-based patterns when using RTEC . For example, while detecting for a meeting we need to firstly find an instance of *greeting* between the two entities. in FlinkCEP this last concept is defined by having an additional component on the beginning of the pattern. Creating a similar pattern using RTEC , requires the inclusion of a completely separate pattern as a prerequisite. Hence, it is expected for the RTEC to require additional computational steps for such cases, as the prerequisite steps should be executed individually.

Furthermore, these types of patterns are harder to visualize using RTEC , as they do not follow the same intuition the system is designed on, as opposed to FlinkCEP . For example, the requirement of the two persons involved in our meeting pattern to be active needs to be expressed using a separate holdsFor predicate and handle the resulting intervals returned appropriately. On FlinkCEP however, we could simply add an extra condition stating that need for the event in question.

5.1.3 Comparison Criteria

In our comparisons we are concerned with two different aspects of the results given and the process they entail. The first focuses the matches themselves, the similarity of the intervals returned by the two systems and the reasons behind possible differences. The second is the performance of each implementation, by means of execution and time specifically focusing on the recognition process for each system.

Quality of the results The first aspect we are interested in, are the results themselves regarding the similarity between the two system's matches. In order to effectively compare the expressiveness of the two systems we analyze the matches returned while working on the same scenarios. The unit that is used in our comparison are the timepoints included in matches returned examining the uncommon segments appearing the two result sets. We study which are unique to the RTEC system and which to FlinkCEP, and evaluate our results. In order to perform this evaluation we decided to use common metrics such as Recall, Precision and F_1 -score. As the intend of this study is to examine the differences between the two patterns and not the evaluation of these patterns as a representation of concepts originating from the data, we chose to consider the results returned by RTEC as Ground Truth on our evaluation methods.

Execution Time Performance We are interested in a comparison of the performance for both systems in terms of execution time. While we examine the full execution time for our patterns, we also focus on the CER process of our implementations. For the purpose of calculating solely the Recognition Time for the FlinkCEP implementation, and taking into consideration that a separate calculation method of the CEP operator is not provided, we calculated the delay for each event because of this process. We achieved this task by storing the current time value before and after the CEP for each event, finding their difference (measured in milliseconds) and adding those differences. Regarding the RTEC system, measuring the recognition time is more straightforward as we are able to find the time lapsed during the execution of the predicates responsible for the CER process. As we discussed, RTEC also allows the use of windows. We conducted separate runs for both datasets using and not using this feature to provide a comprehensive study of RTEC's capabilities. Furthermore, we provide a comparison between the total execution times for the two systems (for both scenarios regarding RTEC's windows). All experiments regarding the Maritime dataset were conducted using a machine with a Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz and 264GB of memory. For the CAVIAR dataset, we ran our experiments on a machine that includes a Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz and 16GB of memory.

5.2 Comparison

At this point we present the results and their evaluation for each dataset. We split the comparison on the *Execution Time Performance* and *Comparison of Resulting Matches* subsections for each dataset, starting with the Maritime and followed by the Surveillance data.

5.2.1 Maritime Dataset

Quality of the results After examining the results of our comparison, we can deduce that in most case the two approaches of the maritime concepts tend to have identical matches. It seems that in most cases the FlinkCEP system can simulate the recognition process of almost every single-valued simple-fluent based Complex Event proposed for the RTEC system.

Table 6: Maritime Accuracy Comparison. Comparing the results of both systems, supposing the RTEC results to be true, and using timepoints as a unit; hence the True Positives occur on both systems, the False Negatives only on the RTEC matches and vice versa for the False Positives.

Composite Event	Vessels(#)	TP	FN	FP	Precision	Recall	F_1 -Score
withinArea	3185	906241607	167559	1336348	0.999	0.999	0.999
trawlSpeed	260	15590400	87968	2234	0.999	0.994	0.997
trawlingMovement	267	31435979	0	4165	0.999	1.000	0.999
lowSpeed	1192	22299633	158290	0	1.000	0.993	0.996
tuggingSpeed	3241	239435220	491159	4130	0.999	0.998	0.999
sarSpeed	19	2426605	42879	0	1.000	0.983	0.991
changingSpeed	1981	39025964	777035	0	1.000	0.980	0.990

In all pattern cases, the results seem to be close to identical. The timepoints that appear to occur solely on the FlinkCEP resulting set, are caused by the different approaches of the systems for determining the termination timepoint of a pattern. More precisely, suppose that an event occurs on time T_1 ; this event might trigger the termination of a fluent. In this case, RTEC considers the termination point having a timestamp of $T_1 = T_1 + 1$. Furthermore, when dealing with fluents depended on others, this artificial delay can be expanded according to the hierarchy levels defined. On the other hand, in our FlinkCEP implementation we do not take this factor into consideration and expand the resulting match to the maximum on each case, to approximate the RTEC behavior; leading to these differences on the results.

Execution time performance For our time comparison we decided not to use a parallelism factor for the FlinkCEP recognition, to more closely simulate RTEC's execution. Furthermore, our experiments used the Kafka streaming platform⁵ in order to provide the stream onto Flink; we divided our input stream in batches of 1M (10^6) events. For the RTEC system we had two separates runs. On the first one we used non-overlapping

⁵<https://kafka.apache.org/>.

windows with size equal to a day (86,400 seconds). The second run did not include the window feature provided by RTEC , but instead had a run for the full dataset on a single query.

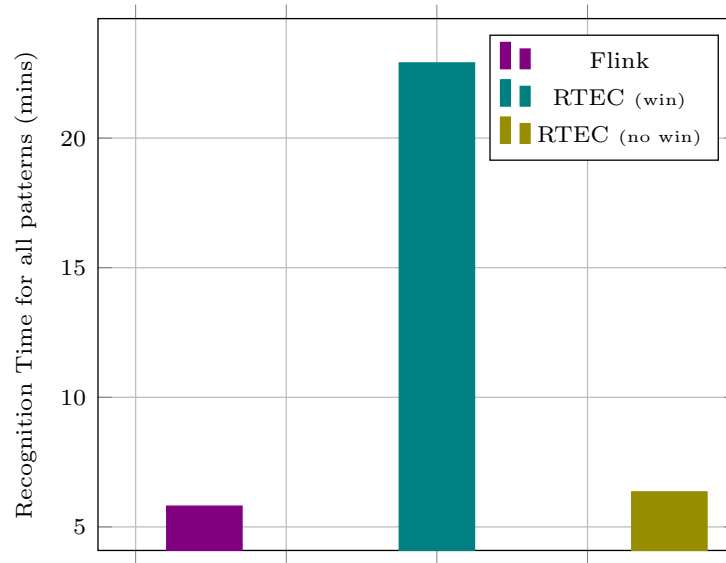


Figure 8: Total recognition time comparison of the two systems (with and without the use of windows in RTEC) for all maritime patterns for the full 6-month Maritime dataset. RTEC is able to detect all patterns passing through the dataset one. FlinkCEP requires separate CER for each pattern; we use the sum of all recognitions.

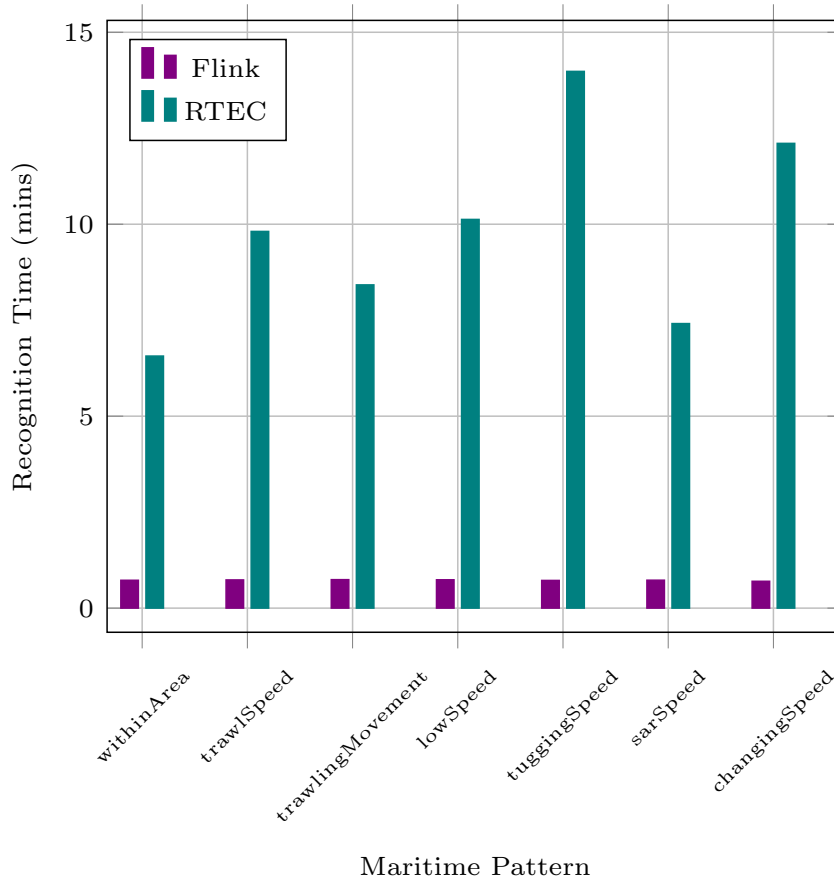


Figure 9: Recognition Time comparison of the two systems for each pattern for the full 6-month Maritime dataset using temporal windows in RTEC.

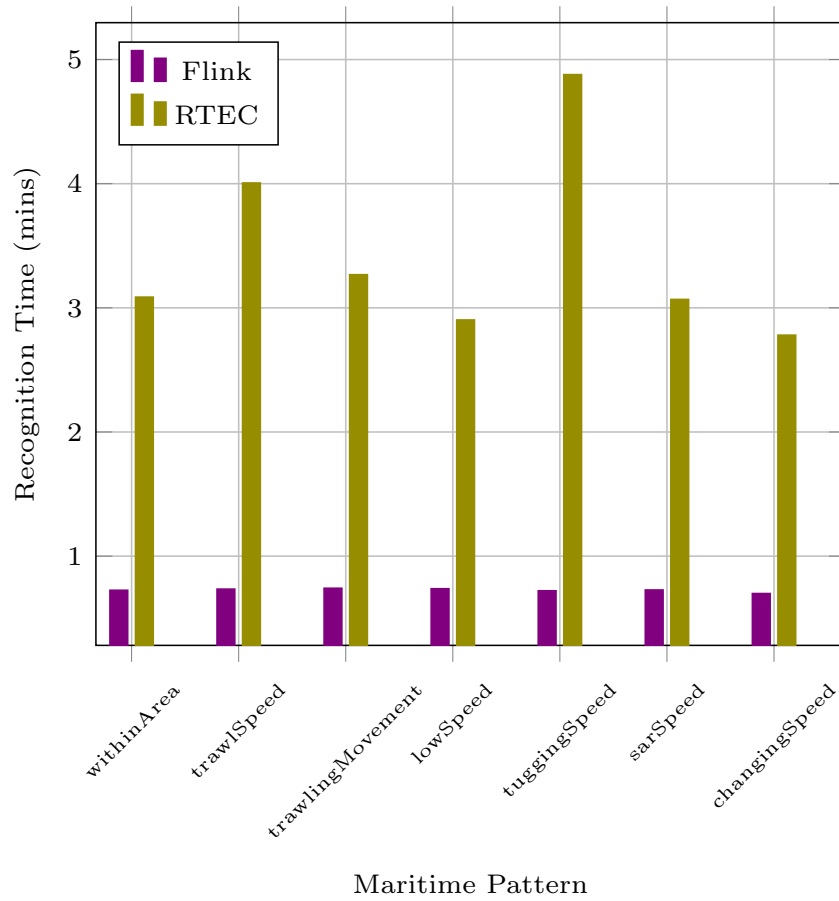


Figure 10: Recognition Time comparison of the two systems for each pattern for the full 6-month Maritime dataset without the use of temporal windows in RTEC.

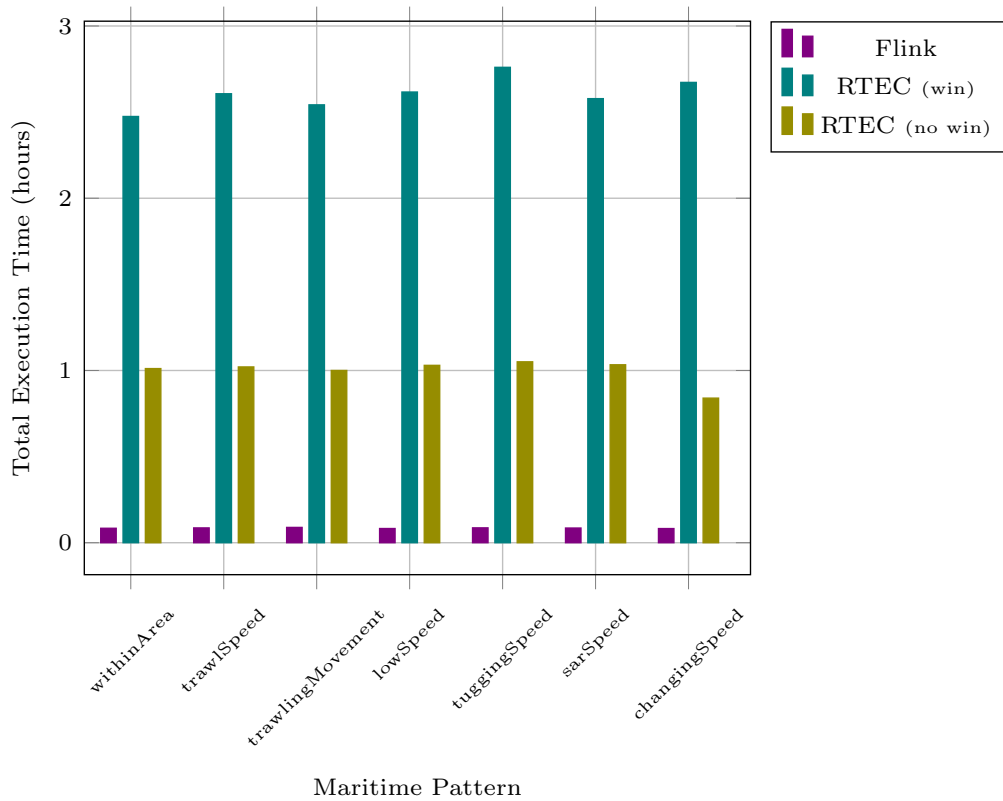


Figure 11: Total execution time comparison of the two systems for each pattern for the full 6-month Maritime dataset.

The above figures indicate that FlinkCEP greatly outperforms RTEC in terms of execution time, for all patterns. As seen by Figure 9 and Figure 10 RTEC requires more time to execute the CER process than FlinkCEP, regardless of the use of windows. Also, we deduce that having temporal windows on our execution does not benefit the performance of RTEC for both the recognition (Figure 8) or the total execution time (Figure 11). Furthermore, we notice that the recognition times for the FlinkCEP implementation do not deviate from a common mean value, regardless of the complexity of the pattern in question, see Figure 9. This leads us to conclude that the performance of the FlinkCEP system is mainly affected by the size of the input stream, rather than the operators within the patterns. Lastly, as seen by Figure 11 the total execution time of the RTEC systems is much greater than the FlinkCEP equivalent, even when the use of windows is not included. This last remark shows that the FlinkCEP system handles large amounts of data efficiently and considerably surpasses RTEC.

As by these results alone we are able to deduce that FlinkCEP is more time-efficient, we decided not to include the execution of Flink with a greater parallelism factor than (1).

5.2.2 Surveillance Dataset

Quality of the results The results, in total, show us that we are able to overcome some of the issues simultaneous events create when using FlinkCEP, by employing the proposed *MegaEvent* structure, and surpass RTEC's efficiency on certain occasions (compared to the ground truth given). The following tables present a similarity evaluation between the two sets of results, along with a comparison to the Ground Truth for each system. The Ground Truth aspect is provided within the CAVIAR dataset, by the use of special annotation.

Table 7: Similarity Comparison for Surveillance pattern. We evaluate the results of RTEC compared to FlinkCEP. In order to do so, we chose to use the RTEC implementation as Ground Truth and evaluate the FlinkCEP results correspondingly. We also are using the timepoints returned as units of our comparisons. The stream is being parsed into Keyed streams, for our FlinkCEP pattern to be simpler.

Pattern	Datastream	TP	FN	FP	Precision	Recall	F_1 -Score
meeting	CAVIAR	1897	144	0	1.000	0.929	0.963
	10×CAVIAR	20167	243	0	1.000	0.988	0.994

Table 8: Meeting Accuracy compared to the Ground Truth. We evaluate the results of the RTEC's and the FlinkCEP's implementations compared to the Ground Truth given. The GT corresponds to the 'meeting' value of the Context tag. We also are using the timepoints returned as units of our comparisons. The stream is being parsed into Keyed streams for our FlinkCEP pattern to be simple.

Datastream	CER System	TP	FN	FP	Precision	Recall	F_1 -Score
CAVIAR	RTEC	1388	283	653	0.680	0.831	0.749
	FlinkCEP	1388	283	509	0.732	0.831	0.778
10×CAVIAR	RTEC	13880	2830	6530	0.680	0.831	0.749
	FlinkCEP	13880	2830	6287	0.688	0.831	0.753

The overall differences rely on the fact that the RTEC system includes matches that do not correspond to a full match, as only the initiation rule has been fulfilled by our data stream. As examined in the previous chapter, these types of matches cannot effectively be defined using the FlinkCEP system. In total they comprise the full differences observed onto our results. The fact that in most cases we detect the same intervals as RTEC does proves that the use of *MegaEvents* can be an effective approach to handling simultaneous events.

Execution Time Comparison Below we present the recognition time for FlinkCEP and RTEC implementations for the CAVIAR dataset. We followed the same methods used for the Maritime dataset, using *Kafka* for FlinkCEP and non-overlapping sliding windows, with size of (1000) video frames this time. We also conducted a separate RTEC run that does not include windows. On our FlinkCEP execution we used keyed streams of Tuple-Events, based on the ids included on each tuple. In order to create such Tuple-Events the input stream underwent a preprocessing step, prior to the Flink process.

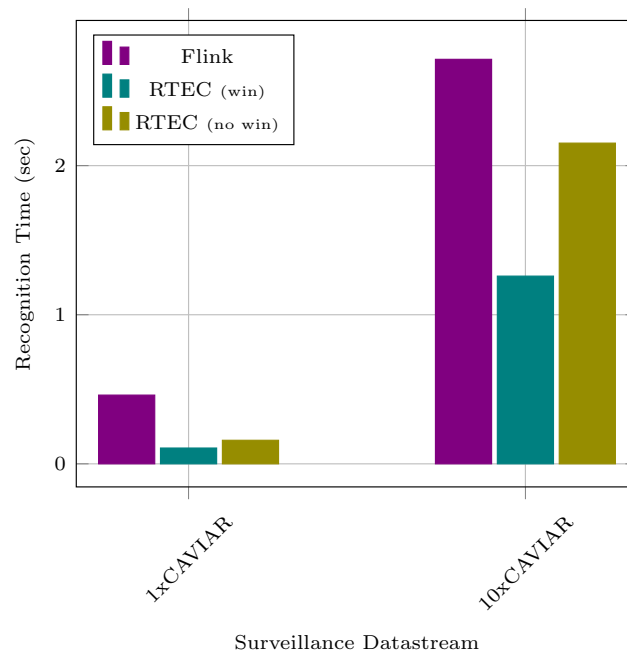


Figure 12: Recognition time comparison of the two systems for each pattern for the ‘meeting’ pattern for the 1×CAVIAR and 10×CAVIAR datasets.

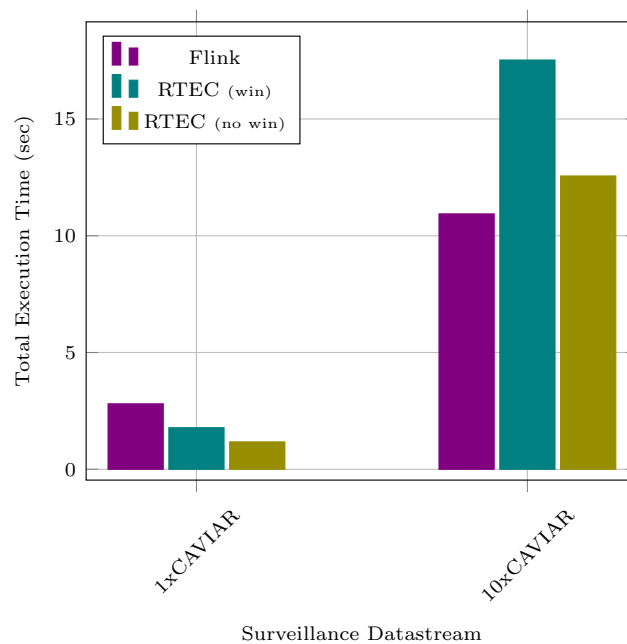


Figure 13: Total execution time comparison of the two systems for each pattern for the ‘meeting’ pattern for the 1×CAVIAR and 10×CAVIAR datasets.

Figure 12 indicates that when dealing with datasets of a smaller size RTEC outperforms FlinkCEP regarding the recognition times. Furthermore, Figure 13 shows that when the stream gets larger, in terms of number of events included, FlinkCEP works more efficiently than RTEC does. As a conclusion we can deduce that RTEC is more appropriate when dealing with small data batches, but suffers in comparison to FlinkCEP when having to handle more than 100k events as input.

5.3 Lessons Learned

After reviewing the results provided by our experiments we come to several conclusions. Some of these conclusions confirm the remarks of the theoretical comparison of the previous chapter.

- **Pattern Hierarchies** We are able to effectively simulate hierarchies between patterns when there is no negation included within their conditions. this can be achieved by incorporating all events of the patterns involved into a single FlinkCEP Complex Event.
- **Unbounded intervals** FlinkCEP proves to be unable to handle a pattern that includes uncompleted matches on their results, because of reaching the end of the stream. The use of such matches would require a different pattern, and thus a different CER process.
- **MegaEvents** We have proven that we are capable of producing MegaEvent types for FlinkCEP , in order to handle simultaneous events and multi-entity patterns. This approach requires an extra preprocessing step and may lead to exponentially large streams, but is effective when the number of events per timepoint is limited.
- **Recognition Time** The results regarding the recognition times of both systems, show that in most cases the FlinkCEP system appears to outperform RTEC . Furthermore, the comparison of the execution times proves that the FlinkCEP system handles the load of data much more efficiently, mostly thanks to the Flink system, resulting on the gap between their performances. Conversely experiments has shown that RTEC works more efficiently when dealing with small amount of data, in terms of input events.

6. CONCLUSIONS AND FURTHER WORK

In this chapter we provide a summary of our study, describe our conclusions and list potential aspects for our comparison that should be extended in future work.

6.1 Conclusions

As we stressed throughout this study, there seems to be several differences on what each FlinkCEP and RTEC are capable of representing efficiently. These differences lay on the disparate approaches they use in order to create and track the same concepts onto a stream of data. The main cause for these differences is the capabilities of each system regarding queries during the recognition process. The fact that RTEC is able to access events without considering them part of the current match allows the design of simple yet expressive patterns, in contrast to the FlinkCEP equivalents. Furthermore, the behavior towards simultaneous events by each system plays a major role on how our patterns are designed, as the order of simultaneous events doesn't affect the RTEC implementations whilst the FlinkCEP system emphasizes on its significance.

Another major contrast between the two systems occur when dealing with matches whose endpoint does not appear on the stream. The reason behind this, is the fact that a complete FlinkCEP pattern would include an *ending* event as part of its declaration; in contrast the RTEC matches might also include ones that only the initiation rule has been triggered, and thus having intervals with no closed end. Moreover, attempting to simulate such behavior with FlinkCEP seems to be more than tricky.

As expected when dealing with any Logic Programming-based system, the existence of the negation creates significant issues. While, the negation on RTEC is achieved by the notion of *negation as failure*, a simple omission of an event or a fact on our FlinkCEP translated patterns does not qualify as a scenario for providing the same results. The main reason that causes that effect emanates from the query capabilities of the two systems, as noted above.

Also, while RTEC relies on the use of windows to achieve a high performance rate, FlinkCEP does not include a window mechanism, even if Flink does provides an implementation regarding other operators. The RTEC system uses sliding (overlapping or not) windows to improve its performance. On the other hand, the only time-related option provided by FlinkCEP (the *within* function) does not simulate the scope of windows and works as a simple condition regarding the first and last components of the pattern's match.

Our experimental results emphasize the importance of the pattern-hierarchy available on RTEC. After looking at the behavior of both patterns with and without the use of hierarchies on their elements and between prerequisites, its necessity is undeniable. And while one might attempt to simulate RTEC's approach in prioritizing and recognizing the prerequisite patterns first and afterwards acting on the resulting stream, we must consider that usually patterns involve information of the original stream as well as the new data created by another recognition process.

Finally, the execution time comparison, focusing on the recognition process, shows that FlinkCEP performs more efficiently than RTEC. The reason these resulting times are so different seems to be caused by the use of Flink and its impressive performance compared

to Logic Programming systems.

6.2 Future work

Several aspects of our work can be extended in order to expand our study in the future. The main directions that such future work may follow are:

- **Statically Determined Fluents** Expand our work by including Statically Determined Fluents of RTEC and presenting implementations using FlinkCEP for such patterns. Even though several of our experiments (not presented in this study) have proven that some SDFs can be approached effectively, we need to examine all possible scenarios. A different approach would be by strictly proving the equivalence of every SDF to a Simple Fluent and thus focusing on this comparison.
- **Multi-valued Fluents** In this study we focused on fluents that can only carry two types of values; we called these fluents Boolean. A significant expansion would be examining whether Fluents that can carry a wider range of different values can be translated as FlinkCEP patterns. These types of fluents are determined by separate rules for each of their values, combined with the property that it can only carry a single value at any given moment (or not carry any value), thus differentiating such a study from ours.
- **RTEC -2** Furthermore, a version of RTEC that handles recursive definitions of Complex Events and long-term relations is under development [46]. This update of the RTEC engine would enable the detection of even more complex patterns, creating the need of a separate study, focused on them.
- **Parallelism** The examination of how parallelism affects the efficiency of our systems can be examined in a further work. In this study, we only provided results without including executions in parallel, but the enabling of parallelism for the RTEC engine should be studied, and more importantly the utilization of such capabilities for an engine based on Flink would provide a thorough comparison of state-of-the-art CER tools.
- **Windows for FlinkCEP** As mentioned, the RTEC system benefits greatly from the use of temporal sliding windows. Even if the Flink does provide a windowing mechanism, combining its use with FlinkCEP is a completely different matter. The examination on how windows can be integrated into the recognition process, together with the use of parallelism, would affect the performance of the FlinkCEP implementation and as a result the comparison.

ACRONYMS

CE	Complex Event
CEP	Complex Event Processing
CER	Complex Event Recognition
FSA	Finite State Automata
RTEC	Run-time Event Calculus
SDE	Simple Derived Events
SDF	Statically Determined Fluent

REFERENCES

- [1] Esper - espertech. <http://www.espertech.com/esper>. Online; accessed: 23-October-2019.
- [2] Asaf Adi and Opher Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004.
- [3] Elias Alevizos and Alexander Artikis. Being logical or going with the flow? A comparison of complex event processing systems. In *Artificial Intelligence: Methods and Applications - 8th Hellenic Conference on AI, SETN 2014, Ioannina, Greece, May 15-17, 2014. Proceedings*, pages 460–474, 2014.
- [4] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. Probabilistic complex event recognition: A survey. *ACM Comput. Surv.*, 50(5):71:1–71:31, 2017.
- [5] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. A logic programming approach to activity recognition. In *Proceedings of the 2nd ACM international workshop on Events in multimedia, EIMM 2010, Firenze, Italy, October 25 - 29, 2010*, pages 3–8, 2010.
- [6] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. Run-time composite event recognition. pages 69–80, 2012.
- [7] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.*, 27(4):895–908, 2015.
- [8] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. Logic-based event recognition. *Knowledge Eng. Review*, 27(4):469–506, 2012.
- [9] William Brendel, Alan Fern, and Sinisa Todorovic. Probabilistic event logic for interval-based event recognition. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2011, Colorado Springs, CO, USA, 20-25 June 2011*, pages 3329–3336, 2011.
- [10] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker M. White. Cayuga: a high-performance event processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1100–1102, 2007.
- [11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
- [12] Luca Chittaro and Angelo Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12:359–382, 1996.
- [13] Nihan Kesim Cicekli and Ilyas Cicekli. Formalizing the specification and execution of workflows using the event calculus. *Inf. Sci.*, 176(15):2227–2267, 2006.
- [14] Nihan Kesim Cicekli and Yakup Yildirim. Formalizing workflows using the event calculus. In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings*, pages 222–231, 2000.
- [15] Maxime Crochemore and Christophe Hancart. *Automata for Matching Patterns*, pages 399–462. 1997.
- [16] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 50–61, 2010.
- [17] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.
- [18] Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):33:1–33:36, 2018.
- [19] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams. Technical report, 01 2007.
- [20] Christophe Dousson. Extending and unifying chronicle representation with event counters. In *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 257–261, 2002.
- [21] Christophe Dousson and Pierre Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 324–329, 2007.
- [22] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010.
- [23] Nicola Falcionelli, Paolo Sernani, Dagmawi Neway Mekuria, and Aldo Franco Dragoni. An event calculus formalization of timed automata. In *Proceedings of the 1st International Workshop on Real-Time compliant Multi-Agent Systems co-located with the Federated Artificial Intelligence Meeting, Stockholm, Sweden, July 15th, 2018.*, pages 60–76, 2018.
- [24] Robert Fisher. The pets04 surveillance ground-truth data sets. 01 2004.

- [25] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Kamp, and Michael Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217–236, 2017.
- [26] Matthew Fuchs. The event calculus as a programming model for game ai. Technical report, 2009.
- [27] Lajos Fulop, Gabriella Fülöp, Róbert Tóth, János Rácz, Tamás Pánczél, Árpád Gergely, and Árpád Beszédes. Survey on complex event processing and predictive analytics. 08 2010.
- [28] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: a survey. *The VLDB Journal*, pages 1–40, 2019.
- [29] Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, pages 5:1–5:18, 2019.
- [30] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: complex event processing over streams. *CoRR*, abs/cs/0612128, 2006.
- [31] Patrick J. Hayes. The frame problem and related problems in artificial intelligence. Technical report, Stanford, CA, USA, 1971.
- [32] Fredrik Heintz. Recognition in the witas uav project a preliminary report. In *Swedish AI Society Workshop (SAIS2001)*, 2001.
- [33] Elena Ikononovska and Mariano Zelke. Algorithmic techniques for processing data streams. In *Data Exchange, Integration, and Streams*, pages 237–274, 2013.
- [34] Ilya Kolchinsky and Assaf Schuster. Real-time multi-pattern detection over event streams. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 589–606, 2019.
- [35] Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
- [36] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.
- [37] Yuan Mei and Samuel Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 193–206, 2009.
- [38] Adrian Paschke and Alexander Kozlenkov. Rule-based event processing and reaction rules. In *Rule Interchange and Applications, International Symposium, RuleML 2009, Las Vegas, Nevada, USA, November 5-7, 2009. Proceedings*, pages 53–66, 2009.
- [39] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Joussemme. Composite event recognition for maritime monitoring. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019.*, pages 163–174, 2019.
- [40] Cyril RAY, Richard DRÉO, Elena CAMOSSI, and Anne-Laure JOUSSELMÉ. Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, February 2018.
- [41] Nicolo Rivetti, Yann Busnel, and Avigdor Gal. Flinkman: Anomaly detection in manufacturing equipment with apache flink: Grand challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 274–279, 2017.
- [42] Georgios M. Santipantakis, Akrivi Vlachou, Christos Doulkeridis, Alexander Artikis, Ioannis Kontopoulos, and George A. Vouros. A stream reasoning system for maritime monitoring. In *25th International Symposium on Temporal Representation and Reasoning, TIME 2018, Warsaw, Poland, October 15-17, 2018*, pages 20:1–20:17, 2018.
- [43] Murray Shanahan. *Solving the frame problem - a mathematical investigation of the common sense law of inertia*. MIT Press, 1997.
- [44] Young Chol Song, Henry A. Kautz, James F. Allen, Mary D. Swift, Yuncheng Li, Jiebo Luo, and Ce Zhang. A markov logic framework for recognizing complex events from multimodal data. In *2013 International Conference on Multimodal Interaction, ICMI '13, Sydney, NSW, Australia, December 9-13, 2013*, pages 141–148, 2013.
- [45] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: a second look at complex event processing architectures. In *Proceedings of the 2011 ACM SC Workshop on Gateway Computing Environments, GCE 2011, Seattle, WA, USA, November 18, 2011*, pages 43–50, 2011.
- [46] 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 2019, Darmstadt, Germany, June 24-28, 2019.*, pages 79–90, 2019.

- [47] Martín Ugarte and Stijn Vansummeren. On the difference between complex event processing and dynamic query evaluation. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.