TECHNICAL UNIVERSITY OF CRETE

MASTER THESIS

A Streaming Implementation of the Event Calculus

Author:

Alexandros MAVROMMATIS

Supervisor:

Prof. Minos GAROFALAKIS

THESIS COMMITTEE

Prof. Minos GAROFALAKIS (Technical University of Crete)

Assoc. Prof. Michail G. LAGOUDAKIS (Technical University of Crete)

Dr. Alexander ARTIKIS (University of Piraeus, N.C.S.R. Demokritos)

Technical University of Crete

School of Electronic & Computer Engineering

in collaboration with the

National Centre for Scientific Research "Demokritos"

Institute of Informatics and Telecommunications

April 2016

Abstract

Events provide the fundamental abstraction for representing time-evolving information that may affect situations under certain circumstances. The research domain of Complex Event Recognition focuses on tracking and analysing streams of events, in order to detect event patterns of special significance. The event streams may originate from various sources, such as sensors, video tracking systems, computer networks, etc. Furthermore, the event stream velocity and volume pose significant challenges to event processing systems. We propose dRTEC, an event recognition system that employs the Event Calculus formalism and operates in multiple computer cores for scalable distributed event recognition. We evaluate dRTEC experimentally using two real world applications and we show that it is capable of real-time and efficient event recognition.

Περίληψη

Τα γεγονότα παρέχουν την θεμελιώδη αφαίρεση για την αναπαράσταση μιας χρονικά εξελισσόμενης πληροφορίας, η οποία μπορεί να επηρεάσει καταστάσεις κάτω από ορισμένες συνθήκες. Το πεδίο έρευνας της Αναγνώρισης Σύνθετων Γεγονότων επικεντρώνεται στην παρακολούθηση και ανάλυση ροών από γεγονότα, με στόχο τον εντοπισμό προτύπων από γεγονότα ιδιαίτερης σημασίας. Οι ροές από γεγονότα μπορούν να προέρχονται από διάφορες πηγές, όπως αισθητήρες, συστήματα παρακολούθησης με βίντεο, δίκτυα υπολογιστών κ.λπ. Επιπλέον, η ταχύτητα και ο όγκος των ροών από γεγονότα δημιουργούν σημαντικές προκλήσεις για τα συστήματα επεξεργασίας γεγονότων. Προτείνουμε το dRTEC, ένα σύστημα αναγνώρισης γεγονότων που χρησιμοποιεί το φορμαλισμό του Λογισμού Πράξης και λειτουργεί σε πολλαπλούς πυρήνες υπολογιστών για κλιμακούμενη και κατανεμημένη αναγνώριση γεγονότων. Αξιολογούμε πειραματικά το dRTEC, χρησιμοποιώντας δύο εφαρμογές του πραγματικού κόσμου και προβάλλουμε τη δυνατότητά του για αποδοτική αναγνώριση γεγονότων σε πραγματικό χρόνο.

Acknowledgements

To start with, I would like to thank my advisors, Dr. Georgios Paliouras and Dr. Alexander Artikis, for the opportunity they gave me to complete my master thesis at N.C.S.R. Demokritos. Without their advise and guidance, I would have never been able to succeed in of the most important goals in my life and become familiar with the field of research. Furthermore, I would like to thank my supervisor, prof. Minos Garofalakis, for his unwavering trust in completing a collaborative thesis and gaining experience outside the usual boundaries of a university.

I am also grateful to my colleague Dr. Anastasios Skarlatidis for his assistance and advise for the successful completion of my thesis. With his knowledge and patience, he guided me to bypass basic obstacles and thinking out of the box.

A big thank you to my friends and colleagues, and especially Konstantinos Pechlivanis and Evangelos Michelioudakis, for their assistance and support throughout these two and a half years.

Last but not least, a massive thank you to my family for being there supporting me throughout my effort and encouraging me in any difficulty I faced.

Alexandros Mavrommatis
Chania, April 2016

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

As the number of sources increases which can feed a system with input data, so does the need for distributed systems. These systems can handle big flows of data, since the traditional data processing systems are inadequate. One of the well known challenges of big data processing is the complex event recognition. Event recognition systems aims to detect real-time events from the incoming flow according to specific patterns. Such systems tend to lead to more confident decision making. Better decisions can result in greater operational efficiency, cost reduction and minimized risk.

Event recognition systems receive as input a stream of time-stamped events (SDEs), in order to identify composite events (CEs). A SDE is a low-level event produced by a sensor or device, which may or may not have been preprocessed [25]. A CE is a high-level recognized event which has been derived according to a concrete definition. Such a definition imposes temporal, and possibly, atemporal constraints on its subevents - SDEs or other CEs.

There are numerous distributed event recognition systems in the current literature [15]. Most of these systems are concentrating in representing the complex event definitions in a query model [10, 20, 34]. They exhibit a method for query optimization by translating them into automata and distributing them among a cluster of computers.

We present a distributed streaming engine, including novel techniques for fast CE recognition, called “distributed Event Calculus for Run-Time reasoning” (dRTEC). dRTEC uses an efficient dialect of the Event Calculus [21], a logic programming formalism for representing and reasoning about events. dRTEC includes novel techniques for efficient and real-time CE recognition, scalable to large number of SDEs and CEs, represented as follows:

- A distributed data streaming CE recognition system using Apache Spark Streaming framework, containing a window mechanism and an interval manipulation operation for the CE intervals for efficient reasoning.
- An indexing method which separates SDEs according to their entities for real-time CE recognition.

- An evaluation of two real-world big data applications.

We evaluate dRTEC experimentally using two real-world applications: activity recognition and maritime surveillance. In activity recognition, the SDEs are "short-term activities" detected on video frames such as a person walking, running, moving abruptly or being active or inactive. The goal is to recognize "long-term activities" which are combinations of the "short-term" ones. Such activities are a person leaves an object unattended, two people moving together, fighting or meeting. Maritime surveillance contains SDEs which are critical movements (MEs) detected from vessels sailing through the Greek seas. Such MEs are a vessel starts or ends having low speed, a vessel's speed change, a communication gap starts or ends, and a vessel turns or is stopped. In maritime surveillance, we aim to recognize CEs related to vessel behaviour such as illegal shipping, suspicious vessel delay, and a vessel approaches fast another one.

1.1 Motivation

In recent years, CE recognition systems are being widely used in various applications. They provide a convenient way to detect important incidents and serious hazard, without the human participation. Once the detection is done, they contribute to the decision making operation, in order to prevent or facilitate an imminent situation.

Maritime surveillance application is a fine example of a CE recognition use case. Vessels are equipped with Automatic Identification System (AIS)[1] transponders. These transponders send information about the vessel, such as position, velocity, type, and destination, to other vessels or a base station. This information is fused and then received by the CE recognition system for processing. The detected events are taken into account by the maritime authorities in order to monitor the activities of vessels. Thus, dangerous situations, such as illegal shipping, can be identified and avoided.

In our approach, CE recognition is based on the logic based dialect of Event Calculus. Event Calculus is a notable tool for CE recognition. It has built-in axioms for complex temporal representation, including the formalization of inertia. In addition, Event Calculus contains temporal, as well as atemporal constraints for the CE definitions. There are numerous other approaches, such as [14], [17], [8], [22], which lack the ability of complex reasoning over domain knowledge [7].

On the other hand, our approach is designed for CE recognition over data streams. Several approaches have been published which receive input stream of events, such as [15], [18] and [23]. [29] and [24] handle streams of input events which may be received out-of order and/or with a delay. [22] implements a similar to our window mechanism for processing input data, by simulating sliding windows over the event stream.

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

Nowadays, there is a significant progress around the field of distributed implementations. Novel systems, such as CE recognition systems, process large amount of data. Thus, they tend to be implemented for execution in a cluster of cores. The job is separated in tasks, which will be distributed among the cores, for parallel execution. [34], [33] and [20] achieve this distribution among computers, according to a specific algorithm; either by distributing the CE definitions, or the input data, or the detection system entities (sources/devices).

In conclusion, the motivation of our approach has three basic directions; the implementation of a distributed CE recognition engine among a cluster of computers. This engine should receive a data stream as input, whose events will be represented in an Event Calculus dialect.

1.2 Contributions

In this thesis, we focus on CE recognition over a large amount of data, using an Event Calculus dialect for its representation. The system is distributed for a fast and efficient processing, making the CE recognition accurate in big data applications. In particular the contributions of this thesis are the following:

- A distributed CE recognition system, using an Event Calculus dialect. One of our basic aims is the implementation of a big data application system. The distribution of the tasks among several cores and the parallel execution of them makes CE recognition significantly fast to process such large datasets. Using Event Calculus makes more convenient for the developer to create CE rules, since its syntax is based in natural language and it is domain-independent.
- An indexing method which separates the data and allocates it amongst the cores. This is achieved based on the input entities of the data. That makes CE recognition more efficient than the aforementioned approaches. Additionally, it includes a dynamic grounding method making our system agnostic to the participating entities avoiding redundant computations.
- We used the Apache Spark framework in our implementation. Apache Spark is a reactive programming tool for building scalable and fault-tolerant streaming applications

1.3 Thesis Outline

In Chapter 2, we provide the required background for Apache Spark, the large-data framework that we used for implementing dRTEC. We also make reference to similar

large-data frameworks which currently exist, as well as a comparison of them. We review and discuss the literature around distributed and non-logic based CE recognition systems. Chapter 3 presents the main architecture design and implementation of our approach. We then, in Chapter 4, present the experimental evaluation of dRTEC using a number of different applications. Finally, in Chapter 5, we present the main conclusions of this work along with discussion regarding the open issues and future directions.

2 Background

In this thesis we focus on distributed event recognition using data streams as input. This chapter provides the required background for our work. Starting from the preliminaries, in Section 2.1 we present the technological basis that we used and the purpose of this selection, in order to implement our work. Our implementation make use of the Event Calculus formalism which is briefly presented in Section 2.2. In Section 2.3, we illustrate an example of the window mechanism that we used in our work. Finally in Section 2.4, we present related work in the domain of distributed, streaming, and CE recognition systems.

2.1 Preliminaries

Nowadays, distributing computing frameworks are widely used in most applications; from communication network applications to big data processing ones. In this section, we describe Apache Spark, an efficient distributed streaming platform for data processing, which is used for the dRTEC implementation. Then, we describe similar distributed platforms and finally we state a comparison of them.

2.1.1 Apache Spark

Apache Spark[1] is an open source distributed computing framework. Spark extends the famous MapReduce model, in order to support different types of operations, such as interactive queries and streaming processing. Spark's main purpose is speed, which is important in processing large amount of data. It has been proved that Spark runs programs 100x faster than Hadoop MapReduce in memory and 10x on disk.

One of Spark's features is the ability to run in-memory computations, comparing to most of the other frameworks that run on disk. Additionally, Spark combines these different processing operations, such as SQL and streaming, to produce complex data analysis pipelines. For that purpose, it provides a rich set of high-level tools such as Spark SQL

[1]Apache Spark website: http://spark.apache.org

for SQL and structures data processing, MLlib for machine learning, Spark Streaming for streaming applications and GraphX for graph processing.

Apache Spark provides high-level APIs in Java, Scala, R and Python with rich built-in libraries. It is able to run in different types of clusters including EC2, Hadoop YARN and Apache Mesos. It is compatible with various other frameworks from DBMSs, such as Apache Cassandra and MongoDB, to message brokers, such as Apache Kafka and RabbitMQ.

Spark's main abstraction is a distributed collection called *resilient distributed dataset* (RDD) [37]. RDD enables an efficient data reuse in intermediate results across multiple computations. It is a fault-tolerant and parallel data structure which provides the ability to the user for persisting results in memory. RDDs are distributed across machines via a partitioning method based on a key for each record (e.g. hash or range partitioned). RDD also provides to the user a rich set of operations, called *transformations* (e.g. map, filter, join). Spark supports fault-tolerance by saving in memory the transformations which will take place on an RDD, building its *lineage*, rather than the actual data. RDD contains another set of operations called *actions*, which return a result value to the application or save it to a storage system. Spark computes RDDs, not at the time of their transformations, but lazily the first time they are used in an action, deriving a pipeline of transformations.

Spark Streaming[2] is one of Apache Spark basic extensions. It extends the core of Spark API which allows data engineers and scientists to process large amount of data in real-time from different sources. The stream processing and computing style is based on micro batches. In Spark Streaming, the developer can write streaming jobs in the same way that writes batch jobs in Spark. Thus, it provides the ability of reusing a Spark application for streaming processing with a minimum modification. Spark Streaming is also fault-tolerant by supporting operations over sliding windows of data. It is a stateful stream processing system, since the developer may maintain a state on disk, based on the data window.

Spark Streaming main collection is *discretized stream* (D-Stream) [38]. D-Stream is built on RDDs – the Spark's main core abstraction – and is actually represented as an immutable sequence of them. D-Stream supports streaming computations as a series of short, stateless, and deterministic tasks on small time intervals. It includes the basic RDD operations of transformations (e.g. map, filter) and actions (e.g reduce).

[2]Apache Spark Streaming website: http://spark.apache.org/streaming/

2.1.2 Related Frameworks

In our days, numerous of platforms have been implemented in order to serve the increasing need of distributed streaming applications. Apache Storm[3] is an open source distributed real-time computation framework for processing large amount of high-velocity data. Storm is efficiently fast with the ability of processing over a million of records per second per node of a cluster. It is a reliable processing system for unbounded streams of data, supporting fault-tolerance in the case of every record will be processed "at least once". It is initially written in the Clojure programming language but it is usable in many programming languages.

Storm is a scalable easy to set up and operate framework. An application written in storm is designed as a "topology" in the shape of a directed acyclic graph (DAG), wiring "spouts" – input data streams – and "bolts" – processing and output modules. A topology is a pipeline of transformations acting similarly to a MapReduce job, but with the difference that the data is processed in real-time and not in individual batches. It is used in various types of applications, such as real-time analytics, online machine learning, etc.

One of Storm's basic extension is Trident. Trident is a high-level abstraction for real-time processing of data in batches. It is on top of Storm, providing operations such as map, reduce, join, aggregate, etc., for stateful stream processing with low latency.

Apache Flink[4] is another commonly used open source scalable distributed stream processing platform. Flink is a dataflow engine in a parallel and pipelined manner. It supports streaming computations over bulks/batches with an efficient and low latency processing. Flink provides high-level APIs in Java and Scala, including numerous libraries, such as Gelly – for graph processing, CEP – for complex event processing, and machine learning. It also supports consuming data from reliable sources such as Apache Kafka.

2.1.3 Framework Comparison

The most suitable method to choose the appropriate distributed framework for a specific application is the comparison of them. We selected Apache Spark for our implementation, collecting the benefits and the drawbacks compared to other famous related frameworks.

One of the newest competitors of Spark is Apache Flink. Spark and Flink are both general-purpose stream processing platforms. They are applicable in a wide field of use cases and large data scenarios. They both include streaming processing, graph processing, machine learning and SQL queries extensions. They are capable of running in standalone mode or in a common cluster.

[3]Apache Storm website: http://storm.apache.org/
[4]Apache Flink website: https://flink.apache.org/

Apache Flink is optimized for cyclic or mostly iterative processing by using iterative transformations on its collections. This derives from an optimization of join algorithms, operator chaining, partitioning reusing and sorting. It is also adequately efficient in batch processing but still it handles data streams as true streams. On the other hand, Spark is based on RDDs. These are in-memory data structures, thus giving the capability of big batch calculations. Spark Streaming wraps data streams into micro-batches, making the streaming processing an explicit batch processing. Moreover, while the batch program is running, the data for the next micro-batch is collected. They both have an efficient memory management but Flink relies on an idiomatic API. At the time we made our option, Flick was not quite developed, making Spark a more mature with a bigger community framework. Spark has also the same capabilities with Flink in a micro-batch processing – the purpose of our option.

Apache Storm is the fundamental framework which antagonizes Apache Spark in most cases. Both platforms are distributed stream processing ones, with lots of benefits for efficient computations in big data. They are compatible with Hadoop distributions, numerous cluster managers such as Apache Mesos and Hadoop YARN, as well as DBMSs such as Apache Cassandra and MongoDB. Both frameworks are fault-tolerant and scalable to large volumes of data.

Apache Storm, instead of micro-batching, operates on tuple streams, processing one tuple at a time. Only the Trident extension has the ability of handling streams in micro-batching. It is a lower latency framework comparing to Apache Spark but Spark has the advantage of higher throughput in large datasets. Storm's fault-tolerance is based on the case that every record will be processed "at least once", on the other hand Spark's fault-tolerance is "exactly once". One basic difference between the two platforms is the method used for parallelization; Storm performs task-parallel computations while Spark performs data-parallel ones. This is a more convenient method for our approach. In addition, dRTEC needs the execution of operations of MapReduce model, which are available in Spark. We may have used Storm along with the Trident extension, since it simulates the same behaviour with the one that Spark does. However, Trident was relatively new, at the time we selected the appropriate framework. Spark is also being maintained and updated a lot more than Storm is, over the last years. Finally, dRTEC needs the execution of stateful operations, maintaining an in-memory state for the former window. Storm does not support such operations, thus the developer has to implement them by himself.

2.2 Event Calculus

Our approach make use of a syntax, presented in RTEC [9], for the SDEs and CE definitions representation. RTEC is a CE recognition system using an Event Calculus dialect

(EC). EC [21] is a logic programming formalism for representing and reasoning about events and their effects. The time model of the dialect which is used in RTEC is linear, including integer time-points. Variables start with an upper-case letter, while predicates and constants start with a lower-case letter. We define F as $fluent$ – a property that is allowed to have different values at different time-points – while $F = V$ denotes that fluent F has value V. Boolean fluents also exist whose possible values are $true$ or $false$.

Table 2.1: Main predicates of RTEC.

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

Table 2.1 summarizes the RTEC predicates available to the event description developer. `holdsAt`$(F{=}V, T)$ represents that fluent F has value V at a particular time-point T. `holdsFor`$(F{=}V, I)$ represents that I is the list of maximal intervals for which $F = V$ holds continuously. holdsAt and holdsFor are defined in such a way that, for any fluent F, `holdsAt`$(F{=}V, T)$ is valid if and only if $T \in I$ for which `holdsFor`$(F{=}V, I)$.

The happensAt predicate represents an instance of an event type. E.g. in activity recognition `happensAt`(`appear`$(id0)$, 10) represents the occurrence of event type $appear(id0)$ at time-point 10. When it is clear from the context, there is no need to separate an event from its type. The definitions are based on the use of happensAt predicate, the effects of events with initiatedAt and terminatedAt predicates, and the values of the fluents with the use of holdsAt and holdsFor predicates. There may be also other possibly atemporal constraints, which transcend the boundaries of EC but are applicable in RTEC.

The last four items of Table 2.1 are interval manipulation predicates specific to RTEC. `union_all`$(L,\ I)$ computes the list I of maximal intervals representing the union of maximal intervals of the lists of list L. For example:

`union_all`$([[(10, 20)], [(4, 15), (18, 22), (30, 42)]],\ [(4, 22), (30, 42)])$

An interval of the form (T_s, T_e) in RTEC represents the closed-open interval $[T_s, T_e)$. I in `union_all`$(L,\ I)$ is a list of maximal intervals that includes each time-point that is part of at least one list of L. `intersect_all`$(L,\ I)$ computes the list I of maximal intervals such that I represents the intersection of maximal intervals of the lists of list L. For example:

`intersect_all`$([[(10, 20)], [(4, 15), (18, 22), (30, 42)]],\ [(10, 15), (18, 20)])$

I in `intersect_all`$(L,\ I)$ is a list of maximal intervals that includes each time-point that is part of all lists of L. `relative_complement_all`$(I',\ L,\ I)$ computes the list I of maximal intervals such that I represents the relative complement of the list of maximal intervals I' with respect to the maximal intervals of the lists of list L. For instance:

`relative_complement_all`$([(5, 25), (27, 46)], [[(4, 12)], [(18, 22), (30, 42)]],$
$[(13, 17), (23, 25), (27, 30), (43, 46)])$

I in `relative_complement_all`$(I',\ L,\ I)$ is a list of maximal intervals that includes each time-point of I' that is not part of any list of L. `complement_all`$(I',\ I)$ computes the list I of maximal intervals such that I represents the complement of the list of maximal intervals I' with respect to the current window interval. For example, if current window interval is $(50, 100)$, then:

`complement_all`$([(55, 60), (72, 88), (90, 97)],\ [(50, 55), (60, 72), (88, 90), (97, 100)])$

2.2.1 Simple Fluents

RTEC fluents belong to two categories: *simple* and *statically determined*. We assume, without loss of generality, that these types are disjoint. A fluent F is simple fluent when $F = V$ holds at a particular time-point T, if $F = V$ has been initiated at an earlier time-point and has not been terminated yet. This is an implementation of the law of inertia. For a simple fluent, the list I in `holdsFor`$(F{=}V,\ I)$ contains the maximal intervals at which $F = V$. These intervals are computed by the amalgamation of all the time-points between the time-point at which F initiated having the value V and F terminated having this value. initiatedAt and terminatedAt rules are domain-specific. If $F = V_2$ is initiated at T_f then effectively $F = V_1$ is terminated at T_f, for all other possible values V_1 of F. Thus, a fluent cannot have more than one value at a specific time-point. However, a fluent is not necessary to have a value at every time-point. There is a difference between initiating a boolean fluent $F = false$ and terminating $F = true$; the former implies, but is not implied by the latter.

In activity recognition, we are interested in capturing the activity of leaving an object unattended – consider the following formalization:

$$\begin{aligned}&\texttt{initiatedAt}(\texttt{leaving\_object}(P, Obj) = true, T) \leftarrow \\ &\quad \texttt{happensAt}(\texttt{appear}(Obj),\ T), \\ &\quad \texttt{holdsAt}(\texttt{inactive}(Obj){=}true,\ T), \\ &\quad \texttt{holdsAt}(\texttt{close}(P,\ Obj){=}true,\ T), \\ &\quad \texttt{holdsAt}(\texttt{person}(P){=}true,\ T)\end{aligned} \tag{2.1}$$

$$\begin{aligned}&\texttt{terminatedAt}(\texttt{leaving\_object}(P, Obj) = true, T) \leftarrow \\ &\quad \texttt{happensAt}(\texttt{disappear}(Obj),\ T)\end{aligned} \tag{2.2}$$

$leaving\_object(P, Obj)$ is a boolean fluent denoting that person P leaves the object Obj unattended. Under certain circumstances, an unattended object may be considered suspicious and therefore we need to record it. $appear$ and $disappear$ are instantaneous SDEs produced by the underlying computer vision algorithms. An entity "appears" when it is first tracked. Similarly, an entity "disappears" when it stops being tracked. An object carried by a person is not tracked–only the person that carries it is tracked. The object will be tracked, that is, it will "appear", if and only if the person leaves it somewhere. $inactive$ is a durative SDE. Objects (as opposed to persons) can exhibit only inactive activity. $close(P, Obj)$ is a statically determined fluent indicating whether the distance between two entities P and Obj, tracked in the surveillance videos, is less than some threshold of pixel positions. $person(P)$ is a simple fluent indicating whether there is sufficient information that entity P is a person as opposed to an object. According to rule (2.1), "leaving object" is initiated when an inactive entity starts being tracked close to a person. Rule (2.2) dictates that "leaving object" stops being recognised when the entity is no longer tracked.

In addition to events constraints, initiatedAt and terminatedAt predicates in the bodies of rules may specify constraints on fluents. $\texttt{start}(F{=}V)$ (resp. $\texttt{end}(F{=}V)$) is a built-in RTEC event taking place at each starting (ending) point of each maximal interval for which $F = V$ holds continuously.

The maximal intervals during which $leaving\_object(P, Obj) = true$ holds continuously are computed using the built-in RTEC predicate holdsFor from rules 2.1 and 2.2. $\texttt{initiatedAt}(F{=}V,\ T)$ does not necessarily imply that $F \neq V$ at T. Similarly, $\texttt{terminatedAt}(F{=}V,\ T)$ does not necessarily imply that $F = V$ at T. Suppose that $F = V$ is initiated at time-points 30, 40, and 55 and terminated at time-points 60 and 70 (and at no other time-points). In that case, $F = V$ holds at all time-points T such that $30 < T \leq 60$.

2.2.2 Statically Determined Fluents

A fluent F is called *statically determined* if the CE description may include domain-specific holdsFor rules, used to define the values of the fluent F in terms of the values of other fluents. Such holdsFor rules use interval manipulation construct (see Table 2.1). In activity recognition, we are interested in identifying whether two people are greeting each other – consider the following formalization:

$$\begin{aligned}
&\texttt{holdsFor}(\texttt{greeting}(P_1, P_2){=}true,\ I) \leftarrow \\
&\quad \texttt{holdsFor}(\texttt{close}(P_1, P_2){=}true,\ I_1), \\
&\quad \texttt{holdsFor}(\texttt{active}(P_1){=}true,\ I_2), \\
&\quad \texttt{holdsFor}(\texttt{inactive}(P_1){=}true,\ I_3), \\
&\quad \texttt{holdsFor}(\texttt{person}(P_1){=}true,\ I_4), \\
&\quad \texttt{intersect_all}([I_3, I_4],\ I_5), \\
&\quad \texttt{union_all}([I_2, I_5],\ I_6), \\
&\quad \texttt{holdsFor}(\texttt{person}(P_2),\ true)I_7, \\
&\quad \texttt{holdsFor}(\texttt{running}(P_2),\ true)I_8, \\
&\quad \texttt{holdsFor}(\texttt{abrupt}(P_2),\ true)I_9, \\
&\quad \texttt{complement_all}(I_7,\ [I_8, I_9])I_{10}, \\
&\quad \texttt{intersect_all}([I_1, I_6, I_{10}],\ I)
\end{aligned} \tag{2.3}$$

A greeting distinguishes meetings from other, related types of interaction. Similar to $inactive$, $active$ (mild body movement without changing location), $running$ and $abrupt$ are durative SDEs produced by the vision algorithms. According to rule 2.3, two tracked entities P_1 and P_2 are said to be greeting, if they are close to each other, P_1 is active or an inactive person, and P_2 is a person that is neither running nor moving abruptly.

The interval manipulation constructs of RTEC support the following type of definition: for all time-points T, $F = V$ holds at T if and only if some boolean combinations of fluent-value pairs holds at T. The use of interval manipulation constructs leads to more concise definitions of the CEs in many cases. In the absence of these constructs, one would have adopt the traditional style of EC representation, by computing all the initiation and termination points of all possible combinations of fluent-value pairs, and then use the domain-independent holdsFor rule to compute the maximal intervals of the CE. Such formalization is much more complex and lower-level than the representation using interval manipulation. In general, interval manipulation constructs may significantly simplify the durative CE definitions.

2.3 Window Mechanism Example

dRTEC performs run-time recognition by computing and storing the maximal intervals of the fluents and time-points in which the events occur. It is implemented using Apache Spark. Spark uses its own mechanism for constructing windows (for further details see Section 2.1.1). The design and implementation of the window mechanism in our work is based on RTEC [9]. CE recognition is executed at specific query times Q_1, Q_2, ... At each query time Q_i only the SDEs that fall within a particular interval – "window" (W) – are taken into consideration. The rest of the SDEs that do not fall within the window are discarded. The reason that we use windows for the CE recognition is the recognition cost. This procedure is based only on the window size and not the entire history of the SDEs from the beginning of the execution. In the case that we used the complete history, the memory capacity would not be adequately sufficient to store all the intervals and time-points. The window size, as well as the temporal distance between two consecutive query times – the "slide" (Q_i - Q_{i-1}) – can be set by the developer. According to the above information, we may observe the following cases:

- $W = Q_i - Q_{i-1}$. In this case, we have consecutive (non-overlapping) windows. If the SDEs arrive in time, there will be no loss of information. But if the SDEs do not arrive in a timely manner, then the SDEs, that took place before Q_i and arrived after it, will be lost. Similarly the SDEs which took place before Q_i and are revised after it, will not be taken into consideration.
- $W > Q_i - Q_{i-1}$. This is the common case with overlapping windows. Since the majority of SDEs arrive with a particular delay in most applications, a window size larger than the slide duration is required. In this case, the effects of the revised SDEs that took place in $(Q_i - W, Q_{i-1}]$, but arrived after Q_{i-1}, are taken into account.

Note that even when the window size is larger than the slide duration, it is plausable that SDEs may be lost. The only way for the data loss possibility to be minimized, is the increase of the window size. However, an increase to the window size may cause a reduction to the CE recognition efficiency. CE recognition efficiency and fault tolerance depends on each specific application differently. In what follows we give an illustrative example and a detailed account of how "windowing" works in RTEC.

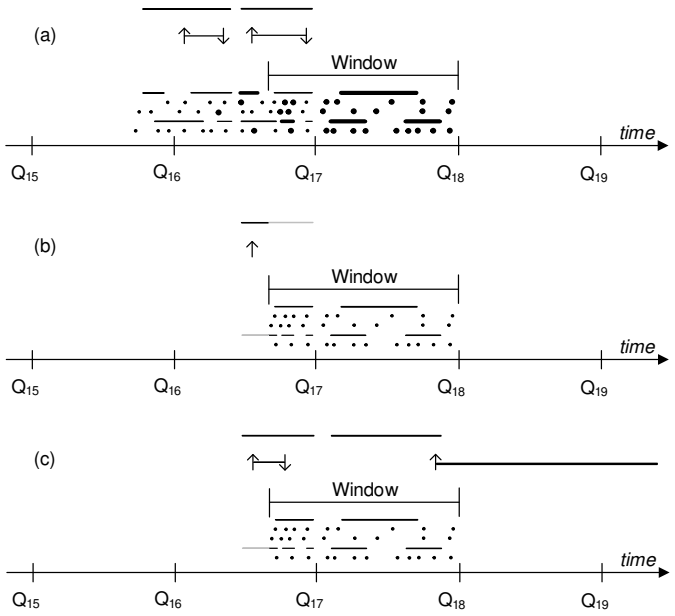


Figure 2.1: Windowing in RTEC.

Figure 2.1 shows windowing in RTEC. To avoid clutter, we present instances of only five SDEs. The instantaneous SDEs are plotted with dots, and durative SDEs with lines. In this example, we have $W > Q_i - Q_{i-1}$; we are interested to recognize two CEs:

- CE_{simple} represented as simple fluent (see Section 2.2.1). Its starting and ending points, as well as its maximal intervals are represented above *Window* in Figure 2.1
- CE_{std} represented as statically determined fluent (see Section 2.2.2). In this example, the maximal intervals of CE_{std} is the result of the union of two durative SDEs. The maximal intervals of CE_{std} are displayed above the maximal intervals of CE_{simple}.

For simplicity, we make the assumption that the maximal intervals of both CEs are defined only by SDEs, and not by other CEs.

Figure 2.1 shows the steps that RTEC follows for CE recognition in a specific window. Figure 2.1(a) displays RTEC before the beginning of CE recognition at query time Q_{18}. The bold dots and lines represent the instantaneous and durative SDEs respectively, which take place between Q_{17} and Q_{18}. This figure also shows the maximal intervals for CE_{std} and CE_{simple} that were computed and stored at query time Q_{17}.

Figure 2.1(b) shows the RTEC state at the beginning of CE recognition at query time Q_{18}. In this window, only the SDEs that take place in $(Q_{18} - W, Q_{18}]$ are taken into consideration. The rest of the SDEs are retracted. The durative SDEs that start before $Q_{18} - W$ and end after it, are partially retracted; only the sub-interval that fall within the window is stored.

CEs will be handled in the same way; the CEs that have been completed before $Q_i - W$ are retracted. The CEs that start before $Q_i - W$ and ends after it, are partially retracted and only a sub-interval which fall within $(Q_i - W, Q_i]$ are taken into consideration. The latter CEs could be also fully retracted, since they have been recognized. The reason, that they are partially retracted, is that they may be revised by SDEs which are received with a delay, or by SDEs which have been erroneous emitted by the sources. A delayed or incorrectly emitted SDE also indicates a delayed or incorrectly recognized CE. There is a case that we could have kept some of the CE intervals and not retract all of them. This is computationally very expensive to determine which of them should be retracted and which not. Thus, the (partial) retraction of all the CE intervals and their computation from scratch for each window, is the only reasonable solution.

According to the above information, Figure 2.1(b) shows the maximal intervals of the two CEs which are recognized, after their retraction. The intervals of CE_{std} that computed at Q_{17} are partially retracted; only the sub-interval which fall within $(Q_{18} - W, Q_{18}]$ is taken into consideration. Similarly for CE_{simple}, the intervals that take place before $Q_{18} - W$ are fully retracted. The starting point of the interval that fall within the window is the only stored; its endpoint has to be recalculated.

Figure 2.1(c) illustrates the step after the CE recognition. As we mentioned above, SDEs which belong in $(Q_{18} - W, Q_{18}]$ are only considered. CE_{std} is defined by the union of two durative SDEs. The new calculated intervals are being amalgamated with the stored intervals from the previous window, in order to create one continuous interval. The first of the maximal intervals happens to be identical to the one which has been calculated in the previous window; there was no revised or retracted SDEs. This is just a feature of the example. If CE_{std} had been defined as the intersection of the durative SDEs, then the intervals would have changed in the current window.

Figure 2.1(c) also shows how the intervals of CE_{simple} are computed at query time Q_{18}. Arrows facing upwards and downwards indicate the starting and ending points of CE_{simple} respectively. Similarly to CE_{std}, the intervals which end before $Q_{18} - W$ are retracted and the starting point of the interval that fall within $(Q_{18} - W, Q_{18}]$ is stored. For simple fluents, it is more convenient to partially retract its intervals, since the starting points are the only ones stored (not an entire sub-intervals). That also requires less memory for each window state than the statically determined fluents. For the current window, we calculate the starting and ending points of the simple fluent using the relevant initiatedAt and terminatedAt rules. Comparing Figure 2.1(a) with 2.1(c), we observe that the ending point of the last interval which was computed at query time Q_{17} was invalidated in the light of the new SDEs that became available at Q_{18}.

Once the starting and ending points have been calculated, we use the domain- independent rule holdsFor, in order to calculate a simple fluent's maximal intervals. The first interval of CE_{simple} became shorter comparing to the one that was computed at the

previous query time. The second interval remains open, since there was no ending point calculated.

The example above shows the CE recognition procedure when SDEs arrive with a variable delay. CE intervals are computed at an earlier query time. Thus, they may be (partially) retracted at the current or a future query time, when SDEs arrive with a delay or are revised. Depending on the application requirements, RTEC may be set to report:

- CEs, the moment when they are recognized, even if they may be (partially) retracted in the future.
- CEs which may partially, but not completely retracted in the future; whose intervals overlap $Q_{i+1} - W$.
- CEs which will not be retracted at all in the future; whose interval end before $Q_{i+1} - W$.

2.4 Related Work

This section provides an overview of previous works, particularly related to complex event recognition. There is a range of different systems that have been used to detect events using various approaches. We outline complex event recognition systems, as well as distributed and streaming systems with similarities to our approach.

2.4.1 Complex Event Recognition Systems

CE recognition systems combine events that occur in an environment, in order to detect composite events. In this section, we present CE recognition systems that have been developed for different application domains. CE recognition systems' primary goal is the fast combination of logical, temporal and spatial constraints to identify CEs accurately.

NagiaraCQ [12] is one of the first widely used CE recognition system. It uses an XML-QL language to represent CE rules in queries. It contains a non-partitioning pattern matching method, using a graph of algebraic operators, in order to detect high level events.

Most of the modern CE recognition systems contain a partitioning or indexing mechanism to achieve the aforementioned goal. *Cayuga* [16] is an example of a high-performance CE recognition system. It represents CE definitions in queries which are later translated into automata, using event algebra extensions. It contains a custom heap management, indexing of operator predicates and reuse of shared automata instances. However, it remains a non-distributed CE recognition system without optimizations, such as query rewriting.

Hirtzel [19] proposes another work where CE definitions are represented as queries which are translated into automata. In this approach, the automata operators are separated into different partitions for parallel execution. Although this method supports stateful operations, it would be a nasty option for a distributed implementation; It would create shuffle dependencies among the cores, causing memory misuse.

Akdere et al. [6] present a complex event detection system in monitoring environment with many distributed sources. This approach uses several operators to combine primitive events in order to produce complex ones. These operators show the temporal correlation among the events. Thus, it creates an event detection graph for each separate complex event, whose leaf nodes represent the primitive events and the non-leaf nodes represent the participating operators. Additionally, using several techniques, it creates plans represented in FSMs. The goal is the best plan selection for the event detection which has the lowest cost-latency ratio, in order to reduce the demand of a continuous data receiving from sources.

Event Calculus [21] is not a query language per se, but it has been widely used to model event querying and reasoning for CE recognition. More recently, variants of Event Calculus has been proposed that extend it, in order to be better suited for CE recognition, such as [31], [9], and [30]. RTEC [9] is an Event Calculus formalism for representing and reasoning about events and their effects. It contains an indexing mechanism that makes it robust to SDEs which are irrelevant to the CEs we want to recognize, as well as a windowing mechanism to support real-time CE recognition.

2.4.2 Distributed and Data Streaming Systems

This section presents several distributed and streaming CE recognition approaches that recognize CEs from heterogeneous streams of information, such as sensor data. The CE definitions are described by patterns that isolate several primitive events and combine their mutual relations. There are numerous distributed CE recognition systems that represent CE definitions into automata. The parallel execution in such systems is mainly based on the distribution of either the automata instances or the automata states, and not on the distribution of the input data.

In the previous section we described implementations whose CE recognition is based on specific cost models [6]. In various works, such as [34] and [33], the aforementioned distribution is based on such cost models. These models are mostly used to create more efficient queries which represent the CE definitions, and to reduce the recognition latency. `NEXT` *CEP* [34] is a distributed CE recognition system that translates event query patterns into automata, in order to distribute them among a cluster of machines. `NEXT` *CEP* aims to achieve a more efficient CE recognition and an optimal query distribution. It uses query optimizations, such as query rewriting, and a particular cost model.

Similar approaches, such as Ahmad et al. [5], propose greedy deployment plans to reduce network bandwidth and communication. Schilling et al. [33] manage the rule distribution by creating a graph connecting the nodes of the processing network. Using a placement algorithm, they decrease the network usage through a cost function.

DistCED [32] is another distributed system that translates queries into automata. In this work, they constructed a specific CE language to represent the queries. Although it does not contain a cost model method, it includes a detection policy awaiting the input events to become stable before feeding them to the system.

RIP [10] is a scalable pattern matching system over event streams. The queries that represent the CE definitions follow the `MATCH-RECOGNIZE` model. It distributes the input events which belong to individual run instances of a query's FSM to different processing units. The run instances are distributed in a round-robin scheduling, since the queries are independent to each other.

Borealis [3] is a streaming system representing event patterns into queries. It contains query optimizers using performance statistics. The queries are distributed on a set of *Aurora* [2] engines for parallel CE recognition. Similarly to Akdere et al. [6] approach, *ZStream* [27] uses tree-based plans to model pattern matching queries. Multiple plans with different costs are assigned to each pattern.

Despite of the efficient and accurate CE recognition of the aforementioned approaches, there is a lack of load balancing on the distribution and fault-tolerance. These features are available in our work due to the participation of Apache Spark and the partitioning method used. *Medusa* [13] uses price-based contracts for dynamic load-balancing in a cluster deployment.

In streaming systems, there is always the possibility that the input events are received with a variable delay from the processing system. Our implementation deals with such case by creating a suitable window mechanism. Mutschler et al. [29] created a distributed approach based on the data sensors. Every node in the network runs the same event processing middleware. A buffering middleware also exists for temporal sorting of the input events, in order to deal with delay issues. The buffer feeds the system with data, which is separated into batches, periodically. It is a fault-tolerant system, since it contains a state recovery with a stream replay method.

SASE+ [4] contains shared match buffer to share partial matches across different FSM runs efficiently. It would benefit from a similar to our implementation's distribution, since it does not use shared memory among the cores.

Recent CE recognition systems make use of frameworks which are based on the MapReduce model for the distribution implementation. Chawda et al. [11] designed a distributed system using the Apache Hadoop MapReduce model, in order to create joins among the query intervals. The queries are mostly based on temporal correlations

among the events comparing to our approach that extends Event Calculus beyond interval correlations. It uses Allen's algebra to define the operations of the intervals, such as partitioning, projecting, splitting and replicating.

Numerous CE recognition systems uses Apache Spark for their distribution. The design of most of these systems is based on lambda architecture, that allows to handle massive quantities of data, combining different frameworks. *KillrWeather* [26] is an in-progress distributed system that integrates Apache Spark, Apache Cassandra, and Apache Kafka for fast streaming computations. It receives time series data as input, in order to recognize CEs regarding to weather information, such as temperature, precipitations, etc.

Stratio Streaming [28] is distributed CE processing engine using Apache Spark Streaming. It allows the creation of streams and queries on the fly, sending large data streams, and building windows over the data. The queries are represented in a simple SQL-like language. It runs several instances of the open source *Siddhi* [36] CEP engine for parallel CE recognition, and uses Apache Kafka for a real-time messaging bus to this end. It contains a convenient API which helps the developer to work with live streams straightaway.

3 Distributed Run-Time Event Recognition

CE recognition systems which support data streams as input should be efficient enough to recognize events in real-time, as well as to scale a large number of SDEs. On the other hand, SDEs might not necessarily arrive at the exact time that they took place. There may be a (variable) delay between the time that the CE recognition system received a SDE and its occurrence time (for a further discussion see [35]). Nonetheless, SDEs could be revised or fully retracted in the future. This a potential case, since a SDE may be erroneous emitted from its source, or it was reported by mistake which is later realised. The SDE revision and rejection is not performed by the CE recognition system, but by the underlying SDE detection system.

In this section we present dRTEC, a distributed implementation of RTEC in Spark Streaming using the Scala programming language. In addition to the optimisation techniques of RTEC, such as windowing, dRTEC supports event recognition using a structured set of operations for distributed reasoning. Figure 3.1 illustrates the basic components of the engine using the activity recognition application. dRTEC accepts SDE streams through MQTT[1], a lightweight publish-subscribe messaging transport. Spark Streaming separates the incoming SDE stream into individual sets, called "micro-batches". The window in dRTEC may contain one or more micro-batches. Each micro-batch may contain events, expressed by happensAt, and fluents, expressed by holdsFor (durative) and holdsAt (instantaneous). For example, according to the SDEs in the first micro-batch shown in Figure 3.1, the entity $id0$ started being tracked–"appeared"–at time/video frame 80. Moreover, the entity $id1$ was running continuously in the interval (90,100).

dRTEC performs various types of processing on the incoming SDE streams. These are presented in the sections that follow. The incoming SDEs are parsed and indexed for data partitioning. Then, various event recognition tasks are distributed in different processing threads. These tasks include fluent processing and window mechanism. Efficient and fast event recognition is achieved through the distribution of the input SDEs amongst the

[1] http://mqtt.org/

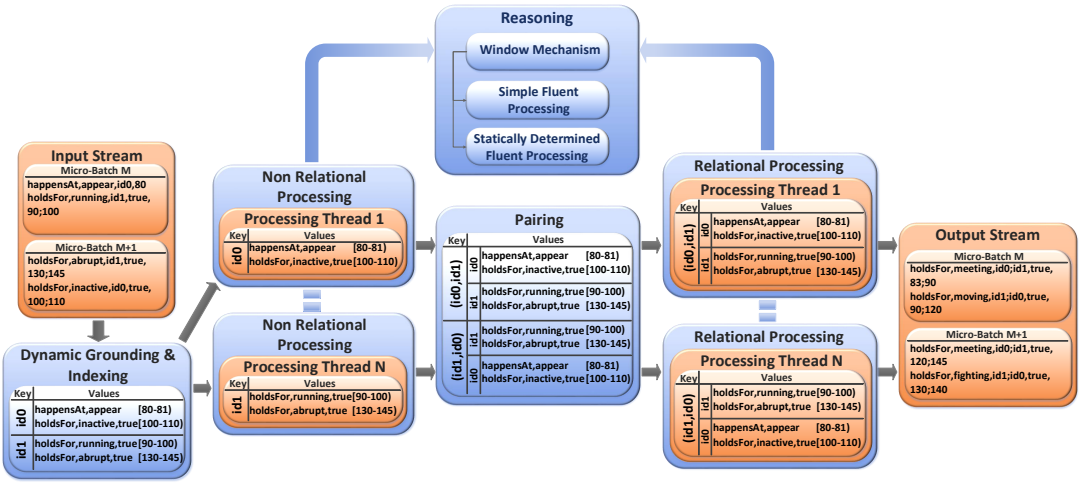


Figure 3.1: dRTEC processing.

processing threads. The CEs that are recognised using the incoming SDEs are streamed out through MQTT (see "Output Stream" in Figure 3.1).

3.1 Dynamic Grounding & Indexing

At each recognition time Q_i, RTEC grounds the CE patterns using a set of constants for the variables appearing in the patterns, except the variables related to time. Moreover, RTEC operates under the assumption that the set of constants is "static", in the sense that it does not change over time, and known in advance. For instance, in the maritime surveillance domain, RTEC operates under the assumption that all vessel ids are known beforehand. Similarly, in activity recognition all ids of the tracked entities are assumed to be known. For many application domains, this assumption is unrealistic. More importantly, there are (many) query times in which RTEC attempts to recognise CEs for (many) constants, for which no information exists in the current window.

To address this issue, dRTEC supports "dynamic grounding". At each query time Q_i, dRTEC scans the SDEs of the current window ω to construct the list of entities for which CE recognition should be performed. Then, it appends to this list all entities that have CE intervals overlapping $Q_i - \omega$. Such intervals may be extended or (partially) retracted, given the information that is available in the current window. In this manner, dRTEC avoids unnecessary calculations by restricting attention to entities for which a CE may be recognised at the current query time.

Indexing is used to convert the input SDEs into a key-value pair format for data partitioning. The partitions are distributed among the available cores/processing threads of the underlying hardware for parallel processing. Each SDE is indexed according to its

entity. In activity recognition, for example, the index concerns the ids of the tracked entities (see "Dynamic Grounding & Indexing" in Figure 3.1). For each window, the SDEs concerning the same entity are grouped together and subsequently sent to the same processing thread. If the number of SDE entities is greater than the available processing threads, the processing threads will contain more than one SDE entity.

3.2 Non-Relational Processing

Indexing is followed by non-relational fluent processing performed at each thread in parallel (see the "Non Relational Processing" boxes of Figure 3.1). Non-relational processing refers to the computation of the maximal intervals of fluents involving a single entity. (In the absence of such fluents, dRTEC proceeds directly to "pairing".) In activity recognition, for example, we want to determine whether a tracked entity is a human or an object (see the rules presented in Section 2.2). An entity is said to be a person if it has exhibited one of the "running", "active", "walking" or "abrupt movement" short-term behaviours since it started being tracked. In other words, the classification of an entity as a person or an object depends only the short-term activities of that entity. The distinction between non-relational and relational processing allows us to trivially parallelise a significant part of the CE recognition process (non-relational CE patterns). The processing threads are independent from one another, avoiding data transfers among them that are very costly.

Non-relational, as well as relational processing, concerns both statically determined and simple fluent processing, and windowing. These tasks are discussed in Section 3.4.

3.3 Pairing & Relational Processing

Relational processing concerns CE patterns that involve two or more entities. In activity recognition, we want to recognise whether two people are moving together or fighting. Prior to relational CE recognition, dRTEC produces all possible relations that may arise from the list of entities computed by the dynamic grounding process–see "Pairing" in Figure 3.1. Then, these relations are distributed to all available processing threads for parallel CE recognition. Note that, in contrast to non-relational processing, the information available to each processing thread is not disjoint. Assume, for example, that the pair ($id0$, $id1$) is processed by processing thread 1, while the pair ($id1$, $id2$) is processed by thread 2. Then both threads will have the output of non-relational processing concerning $id1$ (e.g. the list of maximal intervals during which $id1$ is said to be a person). However, there is no replication of computation, as the output of non-relational processing is cached, and the sets of relations of the processing threads are

disjoint. Furthermore, similar to non-relational processing, each processing thread has all the necessary information, thus avoiding costly data transfers.

3.4 Reasoning

As mentioned earlier, both relational and non-relational processing concern the computation of the list of maximal intervals of fluents. For both types of fluent, simple and statically determined, dRTEC follows the reasoning algorithms of RTEC.

In the windowing mechanism, dRTEC performs run-time recognition by computing and storing the maximal intervals of the fluents and time-points in which events occur. The SDEs and recognized CEs that fall within the window are saved along with their intervals in a Spark Streaming "state" instance. For example, in the case of a simple fluent CE_s, dRTEC checks, at each query time Q_i, if there is a maximal interval of CE_s that overlaps $Q_i{-}\omega$. If there is such an interval then it will be discarded, while its initiating point will be kept. Then, dRTEC computes the initiating points of CE_s in $(Q_i{-}\omega, Q_i]$, and appends them to initiating point (if any) prior to $Q_i{-}\omega$. If the list of initiating points is empty then the empty list of intervals is returned. Otherwise, dRTEC computes the terminating points of CE_s in $(Q_i{-}\omega, Q_i]$, and pairs adjacent initiating and terminating points, as discussed in Section 2.3, to produce the maximal intervals.

Spark Streaming includes its own windowing mechanism. This mechanism is very efficient but lacks of some features that are important for event recognition, such as the determination of the window boundaries. dRTEC windows are temporally defined in order for the "forget" mechanism to take place (retraction of SDE and CE). Spark Streaming does not support a relation between the actual real-time windows and the SDE occurrence times. Thus, an extension to Spark Streaming windowing mechanism was necessary, in order to be functional in our implementation.

Definition 3.1 Initiation of *leaving object* in dRTEC.

$I1 \leftarrow$ GETINTERVAL($occurrences$, Obj, `Fluent`($happensAt$, $appear$))
$I2 \leftarrow$ GETINTERVAL($occurrences$, Obj, `Fluent`($holdsFor$, $inactive$, $true$))
$I3 \leftarrow$ GETINTERVAL($occurrences$, (P, Obj), `Fluent`($holdsFor$, $close$, $true$))
$I4 \leftarrow$ GETINTERVAL($occurrences$, P, `Fluent`($holdsFor$, $person$, $true$))
$I \leftarrow I1$.INTERSECT_ALL($I2$).INTERSECT_ALL($I3$).INTERSECT_ALL($I4$)
AMALGAMATE($occurrences$, (P, Obj), `Fluent`($initiatedAt$, $leaving\_object$, $true$), I)

Definition 3.2 Termination of *leaving object* in dRTEC.

$I \leftarrow$ GETINTERVAL($occurrences$, Obj, `Fluent`($happensAt$, $disappear$))
AMALGAMATE($occurrences$, (P, Obj), `Fluent`($terminatedAt$, $leaving\_object$, $true$), I)

Definitions 3.1 and 3.2 show, respectively, the initiating and terminating conditions of the "leaving object" CE that were presented in Section 2.2.1 in the language of RTEC. Recall that "leaving object" is a simple fluent.

dRTEC uses exclusively intervals in its patterns. The occurrence of an event, such as "appear", is represented by an instantaneous interval. This way, in addition to statically determined fluents, the interval manipulation constructs can be used for specifying simple fluents. In dRTEC, these constructs are supported by "interval instances" and described as follows:

- $interval$.UNION_ALL($list$): produces the union of $interval$ and the list of maximal intervals $list$
- $interval$.INTERSECT_ALL($list$): produces the intersection of $interval$ and the list of maximal intervals $list$
- $interval$.RELATIVE_COMPLEMENT_ALL($list$): produces the relative complement of $interval$ and the list of maximal intervals $list$
- $interval$.COMPLEMENT_ALL(): produces the complement of $interval$ and the current window interval

Algorithm 3.3 GETINTERVAL function.

```
1:  function GETINTERVAL(occurrences, entities, fluent)
2:      if (entities, fluent) ∈ occurrences then
3:          intervals ← occurrences.GET(entities, fluent)
4:          return intervals
5:      else
6:          FLUENTDEFINITION ← definitions.GET(fluent)
7:          FLUENTDEFINITION(occurences, entities, fluent)
8:          intervals ← occurrences.GET(entities, fluent)
9:          return intervals
10:     end if
11: end function
```

The GETINTERVAL function (Algorithm 3.3) retrieves the list of maximal intervals of a fluent. GETINTERVAL has three parameters: (a) the collection "occurrences", i.e. a map pointing to the list of maximal intervals of a fluent, (b) the list of entities of the fluent, and (c) the fluent object. It checks the collection "occurrences" whether the pair $(entities, fluent)$ is already recognized and returns its list of maximal intervals (lines 2-4). If it is not recognized, it retrieves the fluent definition from the list of definitions, and recognizes the fluent (lines 5-9).

Algorithm 3.4 AMALGAMATE function.

```
1: function AMALGAMATE(occurrences, entities, fluent, newInterval)
2:     if (entities, fluent) ∈ occurrences then
3:         intervals ← occurrences.GET(entities, fluent)
4:         ASSERT(occurrences, ((entities, fluent) ⇐ intervals.UNION_ALL(newInterval)))
5:     else
6:         ASSERT(occurrences, ((entities, fluent) ⇐ newInterval))
7:     end if
8: end function
```

The AMALGAMATE function (Algorithm 3.4) is responsible for concatenating a newly calculated interval for a specific pair $(entities, fluent)$ to its list of maximal intervals. AMALGAMATE has four parameters: (a) the collection "occurrences", (b) the list of entities of the fluent, (c) the fluent object, and (d) the newly calculated interval. If the collection "occurrences" already contains the pair, $newInterval$ is concatenated with the previous list of intervals, using the interval manipulation function UNION_ALL (lines 2-4). Otherwise, $newInterval$ is inserted in "occurrences" along with the pair.

We may mention that CE recognition in dRTEC is hierarchical. A CE definition of a higher level uses CE definitions of lower levels, as well as SDEs. Recognition takes place top-down until it reaches CE definition whose recognition is based only on input SDEs (bottom level).

Definition 3.5 Simple fluent *leaving object* in dRTEC.

```
I1 ← GETINTERVAL(occurrences, (P, Obj), Fluent(initiatedAt, leaving_object, true))
I2 ← GETINTERVAL(occurrences, (P, Obj), Fluent(terminatedAt, leaving_object, true))
I ← MAKEINTERVALSFROMPOINTS(I1, I2)
AMALGAMATE(occurrences, (P, Obj), Fluent(holdsFor, leaving_object, true), I)
```

Algorithm 3.6 MAKEINTERVALSFROMPOINTS function.

```
1: function MAKEINTERVALSFROMPOINTS(initPoints, termPoints)
2:     intervals ← [ ]
3:     timepoints ← SORT(initPoints ∪ termPoints)
4:     tempInit ← −1
5:     for all point ∈ timepoints do
6:         if point ∈ initPoints and tempInit == −1 then
7:             tempInit ← point
8:         else if point ∈ termPoints and tempInit ≠ −1 then
9:             ASSERT(intervals, (tempInit, point))
10:            tempInit ← −1
11:        end if
12:    end for
13:    if tempInitiation ≠ −1 then
14:        ASSERT(intervals, (tempInit, ∞))
15:    end if
16: end function
```

The processing of simple fluents is based on the functions described above. Additionally for the simple fluents, we need to calculate their lists of maximal intervals based on their initiation and termination time-points (see Definition 3.5). This is achieved using the MAKEINTERVALSFROMPOINTS function (Algorithm 3.6). For each initiation point, MAKEINTERVALSFROMPOINTS searches for the first termination time-point that follows it. As soon as it is found, a new maximal interval is inserted in the fluent's list (lines 5-12). If there is an initiation time-point left without any termination, then an open interval is added to the fluent's list (lines 13-15).

Statically determined fluents in dRTEC are specified in a similar manner. Definition 3.7, for example, shows the specification of "greeting" (see rule 2.3 for the RTEC representation).

Definition 3.7 Statically determined fluent *greeting* in dRTEC.

$I1 \leftarrow$ GETINTERVAL($occurrences$, $(P1, P2)$, `Fluent`($holdsFor$, $close$, $true$))
$I2 \leftarrow$ GETINTERVAL($occurrences$, $P1$, `Fluent`($holdsFor$, $active$, $true$))
$I3 \leftarrow$ GETINTERVAL($occurrences$, $P1$, `Fluent`($holdsFor$, $inactive$, $true$))
$I4 \leftarrow$ GETINTERVAL($occurrences$, $P1$, `Fluent`($holdsFor$, $person$, $true$))
$I5 \leftarrow I3$.INTERSECT_ALL($I4$)
$I6 \leftarrow I2$.UNION_ALL($I5$)
$I7 \leftarrow$ GETINTERVAL($occurrences$, $P2$, `Fluent`($holdsFor$, $person$, $true$))
$I8 \leftarrow$ GETINTERVAL($occurrences$, $P2$, `Fluent`($holdsFor$, $running$, $true$))
$I9 \leftarrow$ GETINTERVAL($occurrences$, $P2$, `Fluent`($holdsFor$, $abrupt$, $true$))
$I10 \leftarrow I7$.RELATIVE_COMPLEMENT_ALL($I8$.UNION_ALL($I9$))
$I \leftarrow I1$.INTERSECT_ALL($I6$).INTERSECT_ALL($I10$)
AMALGAMATE($occurrences$, (P, Obj), `Fluent`($holdsFor$, $greeting$, $true$), I)

4 Experimental Results

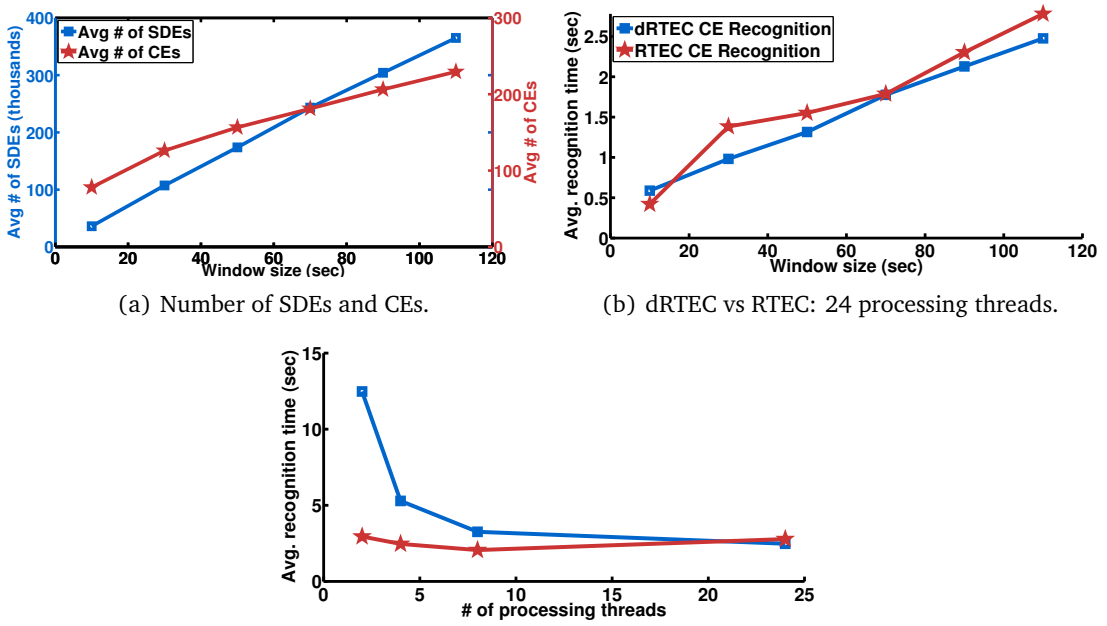
dRTEC has been evaluated in the context of two stream processing systems. In the system of the SYNAISTHISI project, dRTEC is the long-term activity recognition module operating on short-term activities detected on video frames. In the datACRON project, dRTEC recognises suspicious and illegal vessel activities given a compressed vessel position stream produced by a trajectory processing module. The empirical analysis presented below was performed on a computer with dual Intel Xeon E5-2630 processors, amounting to 24 processing threads, and 256GB RAM, running Ubuntu 14.04 LTS 64-Bit with Linux kernel 3.13 and Java OpenJDK 1.8. dRTEC is implemented in Apache Spark Streaming 1.5.2 using Scala 2.11.7. The source code, including the CE patterns for both application domains, is publicly available[1]. dRTEC's warm up period is excluded from the results presented in this section. In all cases, dRTEC recognises the same CEs as RTEC.

4.1 Activity Recognition

The SYNAISTHISI project aims at developing customisable, distributed, low-cost, and automated security and surveillance solutions. To evaluate dRTEC, we used the CAVIAR benchmark dataset[2] which consists of 28 surveillance videos of a public space. The CAVIAR videos show actors which are instructed to carry out several scenarios. Each video has been manually annotated by the CAVIAR team to provide the ground truth for activities which take place on individual video frames. These short-term activities are: entering and exiting the surveillance area, walking, running, moving abruptly, being active and being inactive. We view these activities as SDEs. The CAVIAR team has also annotated the videos with long-term activities: a person leaving an object unattended, people having a meeting, moving together, and fighting. These are the CEs that we want to recognise.

We compare dRTEC results, with the one produced by RTEC, on the discrepancies in the recognised CE time-points. dRTEC has an upper limit that begins to be 100% accurate (window size and slide duration equal to 1,000 ms). Nevertheless, dRTEC is less accurate

[1] https://github.com/blackeye42/dRTEC
[2] http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1



(a) Number of SDEs and CEs.

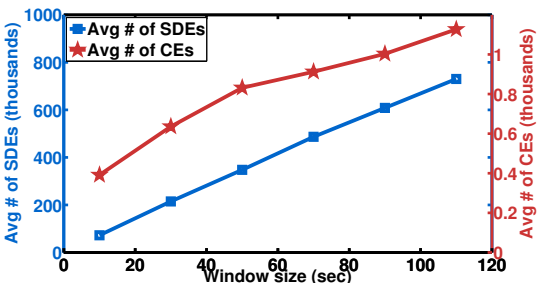
(b) dRTEC vs RTEC: 24 processing threads.

(c) dRTEC vs RTEC: 110 sec window.

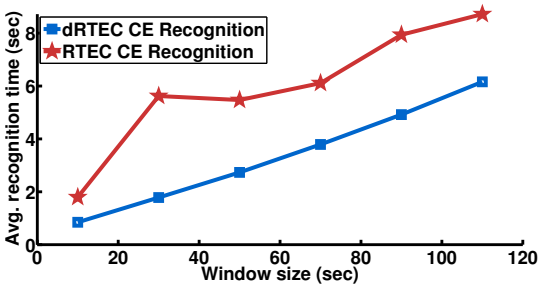
Figure 4.1: Activity recognition with 10 tracked entities.

in the individual executions; there is a big data loss, as well as the CEs which are recognised are rarely correct. The first hypothesis would be that the CAVIAR dataset has a lot of SDEs per window as input, and/or produces a large amount of CEs as output. However we measured the average number of CEs per window which is less than 250 in a window size of 110 sec. The reason which makes dRTEC very inaccurate is that the CEs are correlated to the pairs of the tracked entities. The CAVIAR dataset contains occurrences for 10 entities; that is interpreted to 90 pairs of entities after the pairing operation is done. The procedure for 90 pairs is very slow for such a small number of SDEs as input, making the precise of data receiving infeasible.

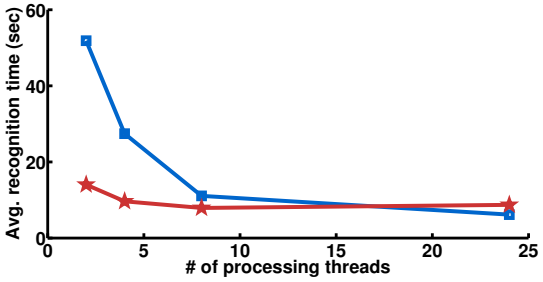
As we mentioned above, the CAVIAR dataset includes 10 tracked entities, i.e. 90 entity pairs (most CEs in this application concern a pair of entities), while the frame rate is 40 milliseconds (ms). On average, 179 SDEs are detected per second (sec). To stress test dRTEC, we constructed a larger dataset. Instead of reporting SDEs every 40 ms, the enlarged dataset provides data in every ms. The SDEs of video frame/time k of the original dataset are copied 39 times for each subsequent ms after time k. The resulting dataset has on average of 3,474 SDEs per sec. Figures 4.1(a)–4.1(c) show the experimental results on this dataset. We varied the window size from 10 sec to 110 sec. The slide step $Q_i - Q_{i-1}$ was set to be equal to the size of the window. Figure 4.1(a) shows the average number of SDEs per window size. The 10 sec window corresponds to approximately 36K SDEs while the 110 sec one corresponds to 365K SDEs. Figure 4.1(a) also shows the number of recognised CEs; these range from 80 to 230. We measured the window state size in memory. It increases as the window size gets larger, but after a given point it remains stable around 300 MB.



(a) Number of SDEs and CEs.



(b) dRTEC vs RTEC: 24 processing threads.



(c) dRTEC vs RTEC: 110 sec window.

Figure 4.2: Activity recognition with 20 tracked entities.

The average CE recognition times per window (in CPU seconds) for both dRTEC and RTEC are shown in Figure 4.1(b). dRTEC made use of all 24 processing threads. To allow for a fair comparison, we invoked 24 instances of RTEC, each using in parallel one processing thread of the underlying hardware. Every RTEC instance was set to perform CE recognition for at most 4 entity pairs, and was provided only with the SDEs concerning the entities of these pairs. For most window sizes, dRTEC outperforms RTEC, but only slightly.

Figure 4.1(c) shows the effect of increasing the number of available processing threads on the performance of dRTEC and RTEC. We varied the number of available threads from 2 to 24; the window size was set to 110 sec. RTEC achieves its best performance early–the increase of processing threads affects it only slightly. In contrast, dRTEC requires all 24 processing threads to match (slightly outperform) RTEC. The cost of data partitioning through dynamic grounding and indexing in dRTEC pays off only in the case of 24 threads.

To stress test further dRTEC, we constructed an even larger dataset by adding a copy of the previous dataset with new identifiers. Thus, the resulting dataset contains a total of 20 tracked entities and 380 entity pairs, while approximately 7K SDEs take place per sec. Figures 4.2(a)–4.2(c) show the experimental results. We varied again the window size from 10 sec to 110 sec. In this case, however, the SDEs range from 72K to 730K (see Figure 4.2(a)). The number of recognised CEs is also much higher; it ranges from 390 to 1100. We also measured the window state size in memory. It has the same behaviour with the one on the previous dataset; after a given point, it remains stable around 700

MB. The larger window state size is expected, since the number of entities in this dataset is twice larger than the previous one, as well as the number of SDEs per sec.

Figure 4.2(b) shows the average CE recognition times per window when all 24 processing threads are available. Each RTEC instance was set to perform CE recognition for at most 16 entity pairs, having available only the SDEs concerning the entities of these pairs. Both dRTEC and RTEC remain real-time, even in the presence of 730K SDE windows. In this set of experiments, dRTEC outperforms RTEC in all window sizes, and the difference is more significant. This is an indication that dRTEC scales better to larger datasets. Figure 4.2(c) shows the effect of increasing the processing threads. We observe a similar pattern to that of the previous experiments (see Figure 4.1(c)).

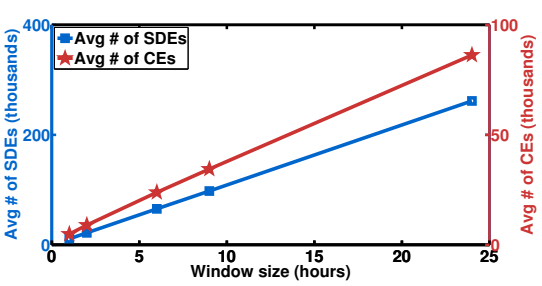
4.2 Maritime Surveillance

The datACRON project targets at introducing novel methods to detect threats and abnormal activity of very large numbers of moving entities in large geographic areas. In the stream processing system of datACRON, dRTEC serves as the component recognising various types of suspicious and illegal vessel activity. We conducted experiments against a real position stream from the Automated Identification System[3], spanning from 1 June 2009 to 31 August 2009, for 6,425 vessels sailing through the Aegean, the Ionian, and part of the Mediterranean Sea[4]. The trajectory detection module of datACRON compresses the vessel position stream to a stream of critical movement events of the following types: "low speed", "speed change", "gap", indicating communication gaps, "turn", and "stopped", indicating that a vessel has stopped in the open sea. Each such event includes the coordinates, speed and heading of the vessel at the time of critical event detection. This way, the SDE stream includes 15,884,253 events. Given this SDE stream, we recognise the following CEs: illegal shipping, suspicious vessel delay and vessel pursuit.

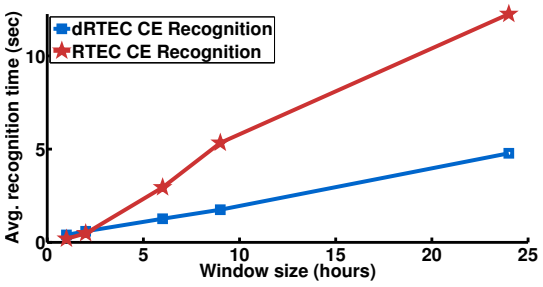
We varied the window size from 1 hour, including approximately 26K SDEs, to 24 hours, including 285K SDEs (see Figure 4.3(a)). The slide step $Q_i - Q_{i-1}$ is always equal to the window size. The number of recognised CEs ranges from 5K to 86K. In other words, the recognised CEs are almost two orders of magnitude more than the CEs in the activity recognition application. As the window size increases, the capacity of memory used for the window state gets larger, since the number of appeared vessels becomes bigger in a larger window. There is a linear relation between the windows size and the size of the window state in memory.

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

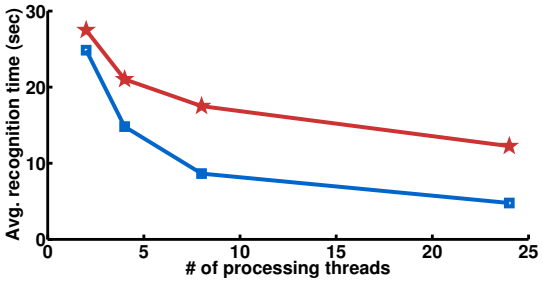
[4] This anonymised dataset (for privacy, each vessel id has been replaced by a sequence number) is publicly available at http://chorochronos.datastories.org/?q=content/imis-3months



(a) Number of SDEs and CEs.



(b) dRTEC vs RTEC: 24 processing threads.



(c) dRTEC vs RTEC: 24 hour window.

Figure 4.3: Event recognition for maritime surveillance.

Figure 4.3(b) shows the average CE recognition times per window when all processing threads are used. Similar to the previous experiments, each RTEC instance was given only the SDEs of the vessels for which it performs CE recognition. Although RTEC matches the performance of dRTEC for small window sizes (1 hour and 2 hour windows), dRTEC scales much better to larger window sizes. In other words, dRTEC seems to perform much better in the presence of a large number of CEs. Figure 4.3(c) shows the effect of increasing the processing threads. Unlike the activity recognition application, dRTEC outperforms RTEC even when just a few processing threads are available. Similar to the activity recognition domain, dRTEC makes better use of the increasing number of threads.

5 Conclusions and Future Work

In this thesis, we focused on implementing a distributed algorithm for event recognition from streaming data. In Section 2.1 we presented Apache Spark – the distributed streaming framework we used for implementing our approach – and we provided information for several similar platforms. Furthermore in Section 2.2, we illustrated the Event Calculus dialect – the logic programming formalism we used for the CE definitions representation. In Section 2.3, we illustrated an example of the window mechanism, in order to make it more comprehensive to the reader. This mechanism faces the problem with the delayed received SDEs. There are numerous CE recognition engines in the recent literature, especially in the field of distributed and data streaming systems. Section 2.4 provides details about these approaches and states the issues which are left open. To address some issues in this thesis, we developed a distributed streaming system using an Event Calculus dialect for the event description.

5.1 Conclusions

In Chapter 3 we described in details our approach along with the novelties which are introduced in the current literature. We analysed the issues that our work deals with and the methods that are used to handle these issues. Finally, we presented our apporach, dRTEC – a distributed Event Calculus for Run-Time Reasoning. We stated its design and architecture, the individual reasoning and partitioning methods that consist it and how they operate.

In Chapter 4 we evaluated dRTEC on two real-world applications. We compared dRTEC with the previous Prolog implementation in the field of velocity and accuracy. Based on the evaluation, we conclude to the following observations:

- dRTEC is a stable and accurate CE recognition system, as soon as there is sufficient time for a window to be established.
- It is a real-time engine regardless the size of the dataset.

- It shows the benefits of a distributed system in large real-word datasets (thousands of SDEs and CEs) comparing to similar non-distributed systems.
- It manages memory efficiently without exceeding its limits and creating bottlenecks.
- dRTEC leverages the available resources of a computer system for fast and efficient CE recognition.

5.2 Future Work

In this section, we describe several directions which can lead to a future extension of our work.

Indexing Optimization

Our work make use of an indexing operation for separating the input data into partitions, and distributing them into a cluster of cores. This indexing operation is based on the input event entity. In most applications, the CE definitions dictate the existence of a relational model amongst the event entities, implemented through the pairing operation. This model is not the most appropriate one for a big number of entities. It creates a large set of partition keys which cannot be executed in parallel, even in a computer with high amount of available resources. An alternative indexing mechanism would be useful to avoid the relational model of the event entities. However, this mechanism should still have the ability to create partitions which are independent to each other, in order to avoid the shuffle dependencies among the partitions. In any other case, the CE recognition would be slow and inefficient.

Event Calculus Prolog Parsing

As we mentioned in previous sections, in our approach we use an Event Calculus dialect for the CE definitions. Event Calculus is a logic programming formalism and is more convenient for the developer to be implemented in a logic programming language, such as Prolog. In our work, we present an Event Calculus representation which is less comprehensive to the developer. The developer should also have an extensive knowledge in the object-oriented programming. Thus, an Event Calculus parser would be useful, converting the Prolog representation into dRTEC representation automatically, without the developer's participation.

Uncertainty

Uncertainty in the CE recognition may be caused by various reasons, such as SDEs that contain noise, erroneous emitted SDEs from the source, incomplete input streams, inconsistent SDE or CE annotation, etc. This issue may lead to inaccurate CE detection,

compromising the performance of a system that contains a CE recognition module. To deal with this issue, we could port dRTEC into probabilistic frameworks.

Bibliography

[1] *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, 2007. www.cidrdb.org.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12 (2):120–139, August 2003. ISSN 1066-8888. doi: 10.1007/s00778-003-0095-z. URL http://dx.doi.org/10.1007/s00778-003-0095-z.

[3] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[4] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376634. URL http://doi.acm.org/10.1145/1376616.1376634.

[5] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004. ISBN 0-12-088469-0. URL http://dl.acm.org/citation.cfm?id=1316689.1316730.

[6] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453869. URL http://dx.doi.org/10.14778/1453856.1453869.

[7] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Real-time complex event recognition and reasoning-a logic programming approach. *Applied*

Artificial Intelligence, 26(1-2):6–57, February 2012. doi: 10.1080/08839514.2012.636616.

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2): 121–142, June 2006. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. URL http://dx.doi.org/10.1007/s00778-004-0147-z.

[9] A. Artikis, M. Sergot, and G. Paliouras. An event calculus for event recognition. *Knowledge and Data Engineering, IEEE Transactions on*, 27(4):895–908, April 2015. ISSN 1041-4347. doi: 10.1109/TKDE.2014.2356476.

[10] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 3–14, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488257. URL http://doi.acm.org/10.1145/2488222.2488257.

[11] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing interval joins on map-reduce. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 463–474. OpenProceedings.org, 2014. doi: 10.5441/002/edbt.2014.42. URL http://dx.doi.org/10.5441/002/edbt.2014.42.

[12] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM. ISBN 1-58113-217-4. doi: 10.1145/342009.335432. URL http://doi.acm.org/10.1145/342009.335432.

[13] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[14] 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 '10, pages 50–61, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-927-5. doi: 10.1145/1827418.1827427. URL http://doi.acm.org/10.1145/1827418.1827427.

[15] 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,

June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677. URL http://doi.acm.org/10.1145/2187671.2187677.

[16] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* DBL [1], pages 412–422. URL http://www.cidrdb.org/cidr2007/papers/cidr07p47.pdf.

[17] Christophe Dousson and Pierre Le Maigat. Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 324–329, 2007.

[18] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: complex event processing over streams (demo). In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings* DBL [1], pages 407–411. URL http://www.cidrdb.org/cidr2007/papers/cidr07p46.pdf.

[19] Martin Hirzel. Partition and compose: parallel complex event processing. In François Bry, Adrian Paschke, Patrick Th. Eugster, Christof Fetzer, and Andreas Behrend, editors, *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 191–200. ACM, 2012. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335506. URL http://doi.acm.org/10.1145/2335484.2335506.

[20] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1315-5. doi: 10.1145/2335484.2335506. URL http://doi.acm.org/10.1145/2335484.2335506.

[21] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

[22] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1): 4:1–4:49, April 2009. ISSN 0362-5915. doi: 10.1145/1508857.1508861. URL http://doi.acm.org/10.1145/1508857.1508861.

[23] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 291–302, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0423-8. doi: 10.1145/2002259.2002297. URL http://doi.acm.org/10.1145/2002259.2002297.

[24] Mo Liu, Ming Li, D. Golovnya, E.A. Rundensteiner, and K. Claypool. Sequence pattern query processing over out-of-order event streams. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 784–795, March 2009. doi: 10.1109/ICDE.2009.95.

[25] D Luckham and R Schulte. EPTS Event Processing Glossary v1.1. Technical report, July 2008.

[26] Patrick McFadin. Stratio Streaming: a new approach to Spark Streaming. Technical report, February 2015. URL http://www.slideshare.net/patrickmcfadin/owning-time-series-with-team-apache-strata-san-jose-2015.

[27] Yuan Mei and Samuel Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 193–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559867. URL http://doi.acm.org/10.1145/1559845.1559867.

[28] David Morales and Oscar Mendez. Owning Time Series with Team Apache: Kafka, Spark Cassandra. Technical report, June 2014. URL http://www.slideshare.net/Stratio/spark-summit-stratio-streaming.

[29] Christopher Mutschler and Michael Philippsen. Adaptive speculative processing of out-of-order event streams. *ACM Trans. Internet Technol.*, 14(1):4:1–4:24, August 2014. ISSN 1533-5399. doi: 10.1145/2633686. URL http://doi.acm.org/10.1145/2633686.

[30] Adrian Paschke. ECA-LP / eca-ruleml: A homogeneous event-condition-action logic programming language. *CoRR*, abs/cs/0609143, 2006. URL http://arxiv.org/abs/cs/0609143.

[31] Adrian Paschke, Alexander Kozlenkov, and Harold Boley. A homogeneous reaction rule language for complex event processing. *CoRR*, abs/1008.0823, 2010. URL http://arxiv.org/abs/1008.0823.

[32] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 62–82, New York, NY, USA, 2003. Springer-Verlag New York, Inc. ISBN 3-540-40317-5. URL http://dl.acm.org/citation.cfm?id=1515915.1515921.

[33] B. Schilling, B. Koldehofe, and K. Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 355–364, Sept 2011. doi: 10.1109/HPCC.2011.53.

[34] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619264. URL http://doi.acm.org/10.1145/1619258.1619264.

[35] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 263–274, New York, NY, USA, 2004. ACM. ISBN 158113858X. doi: 10.1145/1055558.1055596. URL http://doi.acm.org/10.1145/1055558.1055596.

[36] 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 Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1123-6. doi: 10.1145/2110486.2110493. URL http://doi.acm.org/10.1145/2110486.2110493.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.

[38] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522737. URL http://doi.acm.org/10.1145/2517349.2522737.