EFFICIENT EVALUATION OF CORRELATION AND RANKED ENUMERATION FOR COMPLEX EVENT RECOGNITION

ALEJANDRO JOSÉ GREZ ARRUA

Thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the Doctor in Engineering Sciences

Advisor:
CRISTIAN RIVEROS

Santiago de Chile, November, 2023

© MMXX, ALEJANDRO JOSÉ GREZ ARRUA
Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica que acredita al trabajo y a su autor.
EFFICIENT EVALUATION OF CORRELATION AND RANKED ENUMERATION FOR COMPLEX EVENT RECOGNITION

ALEJANDRO JOSÉ GREZ ARRAU

Members of the Committee:
CRISTIAN RIVEROS
DOMAGOJ VRGOC
MARCELO ARENAS
GONZALO NAVARRO
PIERRE BOURHIS
GUSTAVO LAGOS

Thesis submitted to the Office of Graduate Studies in partial fulfillment of the requirements for the Doctor in Engineering Sciences

Santiago de Chile, November, 2023
© MMXX, ALEJANDRO JOSÉ GREZ ARRAU
It has been a long and complex journey to get to this point. Looking back, I think about all the people that some way or another were part of this experience.

I can only begin thanking my wife Rachel. All the hard work and stress of these years was a lot easier having you as my life partner. You helped me going through all the problems, the frustrations, the mind blocks, the lack of motivation. You were also the first one with whom I shared the great moment of joy when I overcame each of them. You always believed in me, and taught me to believe in myself when I thought I couldn’t make it. Rachel, thanks to you, these years I have grown as a person, and I keep doing it each day.

I also want to thank my parents. A lot of the person I am now, with my principles and convictions, is thanks to how you raised me. From the first math games splitting the pizza into fractions, to the physics books and talks about the components of a computer, you always supported my passion for knowledge and gave me the tools and opportunities to get to this point in my life.

I also want to thank the professors that supported my during this time. Specially, I want to thank my supervisor Cristian. Thank you for everything you have taught me and all the opportunities you have given me. You pushed me out of my comfort zone because you knew I could be better. I know I was a complicated case, but you never stopped giving me your support and trust. I also want to thank Pierre Bourhis for having me in Lille and for being a great supervisor and research partner.

Finally, I want to thank the friends and partners that were there during this years, with whom I had lunch, a coffee or maybe a few beers. Life is made of those little moments.

I want to acknowledge the importance of all these people. Thanks to them, this journey has reached its destination.
## Contents

Acknowledgements .............................................. v

List of Figures ............................................... ix

Resumen ....................................................... xi

Abstract ....................................................... xiii

Chapter 1. Introduction ....................................... 1
  1.1. CER query language by example ......................... 2
  1.2. About efficiency in evaluation algorithms ........... 5
  1.3. Hypothesis and research questions .................. 6
    1.3.1. Efficient enumeration algorithms for CER queries with correlation 6
    1.3.2. Efficient ranked enumeration algorithms .......... 6
  1.4. Contributions ........................................ 8
    1.4.1. A formal framework .............................. 8
    1.4.2. Partition-By: the CER correlation operator .... 10
    1.4.3. Ranked enumeration ............................. 11
  1.5. Related Work .......................................... 12
    1.5.1. Formal framework ............................... 12
    1.5.2. Partition-By .................................... 13
    1.5.3. Ranked enumeration ............................. 14
  1.6. Outline .............................................. 15
  1.7. Previous Publications ............................... 16

Chapter 2. Preliminaries ..................................... 18
  2.1. Schemas, tuples and streams ........................ 18
  2.2. CEL syntax and semantics ............................ 19
    2.2.1. Notational conventions .......................... 21
    2.2.2. Unary CEL ..................................... 21
  2.3. Efficiency in Complex Event Recognition ........... 22
<table>
<thead>
<tr>
<th>Chapter 3. A formal framework</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. A Query Language for CER</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1. Discussion of language features.</td>
<td>28</td>
</tr>
<tr>
<td>3.2. A Computational Model for CEL</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1. I/O-deterministic CEA</td>
<td>33</td>
</tr>
<tr>
<td>3.3. Comparing Unary rCEL and CEA</td>
<td>35</td>
</tr>
<tr>
<td>3.4. Selection Strategies</td>
<td>41</td>
</tr>
<tr>
<td>3.4.1. The semantics of common selection strategies in CER</td>
<td>44</td>
</tr>
<tr>
<td>3.4.2. Compiling selection strategies into CEA</td>
<td>48</td>
</tr>
<tr>
<td>3.5. Expressiveness of Fragments of CEL</td>
<td>54</td>
</tr>
<tr>
<td>3.5.1. Unary CEL and the $*$ property</td>
<td>54</td>
</tr>
<tr>
<td>3.5.2. Strict Sequencing versus Strict Selection</td>
<td>62</td>
</tr>
<tr>
<td>3.6. Algorithms for Evaluating CEA</td>
<td>74</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Chapter 4. Partition-By: the CER correlation operator</td>
<td>89</td>
</tr>
<tr>
<td>4.1. Partition-by</td>
<td>90</td>
</tr>
<tr>
<td>4.1.1. Partition-by and hierarchical queries</td>
<td>92</td>
</tr>
<tr>
<td>4.2. Chain Complex Event Automata</td>
<td>96</td>
</tr>
<tr>
<td>4.2.1. I/O-determinization and disequalities</td>
<td>99</td>
</tr>
<tr>
<td>4.3. A Streaming Evaluation Algorithm for Chain-CEA</td>
<td>103</td>
</tr>
<tr>
<td>4.3.1. The run DAG.</td>
<td>103</td>
</tr>
<tr>
<td>4.3.2. An index for binary predicates.</td>
<td>105</td>
</tr>
<tr>
<td>4.3.3. The streaming evaluation algorithm</td>
<td>107</td>
</tr>
<tr>
<td>4.4. A Persistent Index Structure for Equalities and Disequalities</td>
<td>112</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Chapter 5. Ranked enumeration</td>
<td>118</td>
</tr>
<tr>
<td>5.1. Preliminaries</td>
<td>121</td>
</tr>
<tr>
<td>5.1.1. Notation</td>
<td>121</td>
</tr>
<tr>
<td>5.1.2. Efficiency</td>
<td>123</td>
</tr>
<tr>
<td>5.1.3. Ranked enumeration</td>
<td>124</td>
</tr>
<tr>
<td>5.2. MSO Cost Functions</td>
<td>124</td>
</tr>
<tr>
<td>5.3. Applications</td>
<td>134</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A stream $S$ of Twitter events, each with its message, timestamp and position in the stream. $T$ are tweets with their id; $R$ are replies with the id of the tweet they are replying to ($\text{tweetId}$)</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Syntax of CEL. $R$ ranges over relation names, $A$ over variables in $L$, $P(\bar{A})$ over atoms over $P$, and $L$ over subsets of $L$.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>CEL semantics.</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Additional syntax rules of rCEL. $L$ ranges over subsets of $L$.</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>rCEL semantics.</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>A CEA that can generate an unbounded amount of complex events.</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>A CEA for the atomic formula $R$.</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>A stream $S$ of events measuring temperature and humidity. “value” contains degrees and humidity for $T$- and $H$- events, respectively.</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Expressiveness comparison between fragments with different contiguous operators. By Proposition 4, containment $(a)$ holds and is strict for any set of predicates $P$. By proposition 5, $(c)$ holds for any set of predicates $P$. By Theorem 5, $(b)$ holds whenever $P$ contains a binary equality predicate in which case $(c)$ becomes strict. By Proposition 6, $(c)$ becomes $\equiv$ whenever $P$ is a set of unary predicates.</td>
<td>63</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of a Compact Complex Event Set. On the left, we show its structure $\mathcal{D}$ where nodes are labeled with its corresponding event, and the $\text{first}$, $\text{last}$, and $\text{next}$ functions are represented as edges (see node 3 and 6). The directed graphs on the middle and right of the figure are the $\text{Next}<em>{\mathcal{D}}$ and $G</em>{\mathcal{D}}$ of $\mathcal{D}$, respectively.</td>
<td>78</td>
</tr>
<tr>
<td>3.8</td>
<td>A CEA $\mathcal{A}$ in normal form (top left) and stream $S$ (top right) and an illustration of Algorithm 2 on $\mathcal{A}$ and $S$ (bottom figures).</td>
<td>85</td>
</tr>
</tbody>
</table>
4.1 A stream $S$ of events from Twitter. $T$ are tweets with an id, a user-id and a post message, and $R$ are responses with an id, a user-id, a tweet-id, and a reply message. The last line is the index of each event in the stream, respectively. #vote, #ihate, #stop stand for the hashtags #voteforjohn, #ihatejohn and #stophating, in the fictional scenario of US election debates on Twitter.

4.2 An example of chain-CEA with unary predicates $T_1 := \text{type}(T) \land \text{post} = \text{'#vote'}, R_1 := \text{type}(R) \land \text{reply} = \text{'#ihate'},$ and $R_2 := \text{type}(R) \land \text{reply} = \text{'#stop'}.$

4.3 A list of tuples $s_1 \ldots s_7$ with the additional bookkeeping to support disequalities.

5.1 A string-DAG $D$ and the result of $\text{prioritize}(D).$
RESUMEN

Con el paso del tiempo, son cada vez más necesarias herramientas que permitan resolver consultas a datos en tiempo real, lo que se vuelve más complejo a medida que la cantidad de datos que se procesan es vuelve cada vez mayor. El área de Complex Event Recognition (CER) engloba herramientas que buscan solventar esta necesidad, al proveer sistemas particulares especializadas en la evaluación de consultas sobre flujos de datos, enfocándose principalmente en dar respuestas en tiempo real a consultas con un alto nivel de expresividad.

En este trabajo buscamos aportar a esta área al abstraernos de los sistemas desarrollados y estudiar las necesidades más recurrentes de los usuarios de estas herramientas desde un punto de vista teórico. Primero, proponemos un marco teórico para CER, que define un lenguaje básico de consultas con una semántica clara de sus operadores y capaz de expresar el llamado fragmento regular de los lenguajes CER, junto con algoritmos de evaluación que entregan sólidas garantías de eficiencia al usuario: procesamiento de cada evento en tiempo constante y enumeración de cada resultado en tiempo lineal en el tamaño de este.

Luego, nos enfocamos en extender dicho marco teórico de dos maneras. Primero, extendemos el lenguaje con el operador partition-by, que permite expresar una versión restringida de correlación con igualdad e inigualdad, y proponemos un nuevo nuevo algoritmo que permite evaluar consultas con este operador, manteniendo las mismas garantías de eficiencia. Finalmente, proponemos técnicas de evaluación de consultas sobre lógica monádica de segundo orden que permiten entregar los resultados en orden de acuerdo a una función de costos definida por el usuario, que toma tiempo de procesamiento lineal sobre el largo del input y tiene un factor logarítmico del largo del input en el tiempo de enumeración de cada resultado. Luego, utilizamos esta técnica para extender el marco CER propuesto con el operador de ventanas de tiempo within.
Palabras Claves: Procesamiento de Eventos Complejos, Enumeración con delay output-lineal, Enumeración con delay logarítmico, Correlación, Ventanas de tiempo.
ABSTRACT

With the passing of time, the need for tools to answer queries over data in real
time is increasing, and it becomes more complex as the amount of data to be processed
becomes greater. The area of Complex Event Recognition (CER) includes tools that
seek to solve this need, by providing particular systems specialized in the evaluation
of queries over data streams, focussing mainly on providing answers in real time to
queries with high expressiveness.

In this work we aim to contribute to this area by abstracting ourselves from cur-
rently developed systems and studying the more recurrent needs that the users of these
tools have from a theoretical point of view. First, we propose a theoretical framework
for CER, which defines a basic query language with clear semantics of its operators
and capable of expressing the so-called regular fragment of CER languages, together
with evaluation algorithms that provide solid efficiency guarantees to the user: pro-
cessing each event in constant time and enumerating each result in time linear over its
size.

Then, we focus on extending said framework in two manners. First, we extend the
language with the partition-by operator, which allows to express a restricted version
of correlation with equalities and disequalities, and propose a new algorithm able to
evaluate queries with this operator while maintaining the same efficiency guarantees.
Finally, we propose a query evaluation technique over monadic second order logic that
allows to enumerate the results in order according to a user-defined cost function, tak-
ing linear preprocessing time over the input length and having a logarithmic factor over
the input length in the enumeration time of each result. Then, we use this technique to
extend the proposed CER framework with the time window operator within.

Keywords: Complex Event Processing, Output-linear delay enumeration, Logarithmic delay enumeration, Correlation, Time windows.

xiii
Chapter 1. INTRODUCTION

The paradigm of Complex Event Processing has established in various application fields. Events representing changes of state are being fed to the engine, which process them to retrieve relevant data. Complex Event Recognition focuses on the identification of occurrences of patterns within the stream of events. Usually, this recognition is expected to be done in real time, in order to take reactive measures to the complex event occurrence. This, combined with the assumption that many events might arrive to the engine within a small period of time, makes the efficiency of the engine a crucial aspect for it to be useful in practical scenarios. CER has been successfully applied in scenarios like maritime monitoring (Pitsikalis et al., 2019), network intrusion detection (Mukherjee, Heberlein, & Levitt, 1994), industrial control systems (Groover, 2007) and real-time Analytics (Sahay & Ranjan, 2008).

There are prominent examples of CER engines both from academia and industry, which include SASE (Wu, Diao, & Rizvi, 2006), EsperTech (Esper Enterprise Edition website, n.d.), Cayuga (Demers, Gehrke, Hong, Riedewald, & White, 2006), and TESLA/T-Rex (Cugola & Margara, 2010, 2012a), among others; see (Cugola & Margara, 2012b; Giatrakos, Alevizos, Artikis, Deligiannakis, & Garofalakis, 2020; Alevizos, Skarlatidis, Artikis, & Paliouras, 2017) for a survey. They have focussed on practical issues like scalability, fault tolerance, and distribution, with the objective of making CER engines applicable to real-life scenarios. Other design decisions like, for example, the query language used to specify complex event patterns, have received less attention. Often, the language is retro-engineered to match computational algorithms that are used to process data efficiently; see for example (Zhang, Diao, & Immerman, 2014). This work takes a step back and attempts to make a study CER focusing on efficiency, but with solid theoretical foundations.

Two features of CER form the main motivation for this work: correlation and event/time windows. Usually, CER engines provide a query language to the user with which they can express the pattern to look for. Within this language, the correlation operator is used to express that, inside the pattern, two or more events are bounded
to satisfy a certain property (e.g., that two different events that are part of the pattern must have the same value). On the other hand, the window operator is used to say that we only care about the pattern occurrence if it occurs within a bounded period of time/number of events in the stream (e.g., that the pattern occurs within a time lapse of 10 minutes).

In the following work we will study the problem of including this operators in a CER engine. First, we provide a formalization of CER features by analysing the basic operators that are present in most CER query languages, studying their expressiveness, and finally giving evaluation algorithms for them with solid efficiency guarantees; consequently, we end up with a solid CER framework over which we can work. We then extend this framework in two different branches. On one hand, we will adapt the framework to fit a new correlation operator, called partition-by. This operator can be used to add equality constraints among events of the searched pattern, but in a limited way that allows the efficient evaluation of it. On the other hand, we provide a general technique that can be used to retrieve the pattern occurrences in a certain order, defined by the user. Among other applications, this technique will solve the evaluation of the window operator, by retrieving the results ordered by their window size. Both extensions are done maintaining the focus on both efficiency and formal semantics.

Before presenting the hypotheses and research questions that motivate our work, we first present some examples in order to provide a quick insight to our framework.

### 1.1. CER query language by example

To help on the explanation, we find useful to present a running example: the case of the Twitter stream. The data stream of Twitter can be seen as a temporal sequence of events of two different types, \( T(id, \text{timestamp}, \text{msg}) \) (for Tweet) and \( R(id, \text{timestamp}, \text{msg}, \text{tweetId}) \) (for Reply), where each event contains relevant information like the id, timestamp and its message msg (with the hashtags that it contains), and the events of type Reply have an additional field tweetId with the id of the tweet it is replying to. Figure 1.1 depicts such a stream: each column is an event and each row has the value of the corresponding attribute of that event.
In CER engines, the user declares temporal patterns with a high-level, declarative language, and then the engine is the one in charge of recognizing in real time the occurrences of the pattern. For instance, the user might like to detect a pattern, or complex event, of the form:

“A tweet with the hashtag ‘#voteTrump’, followed by a reply with ‘#iHateTrump’”

that might be useful for a journalist that is searching for debates about the USA elections. Let us intuitively explain how we can express this as a pattern (also called a formula):

\[
\varphi_1 = (T; R) \text{FILTER} (T.\text{msg} = '\%\#voteTrump\%') \land R.\text{msg} = '\%\#iHateTrump\%')
\]

This formula is asking for two events, one of type Tweet (T) and one of type Reply (R), and the two events are filtered to select only those representing a tweet and reply with the required hashtags. Here, the notation ‘\%\#w\%’ is used to filter events that contain the substring \(w\). Event streams are typically noisy, and therefore one cannot expect in general that the \(T\) and \(R\) events of \(\varphi_1\) occur contiguously, i.e., next to each other in the stream. For this reason, CER engines allow dismissing irrelevant events. Consequently, in our framework the semantics of the non-contiguous sequencing operator (\(;\);) allows for arbitrary events to occur in between the events of interest, in this case \(T\) and \(R\).

Whenever a pattern is detected in a stream, a corresponding complex event will be output. In this work, each complex event is represented by set of indices (positions in the stream) of the events that witness the matching of a formula. Specifically, let \(S[i]\)
be the event at position \(i\) of the stream \(S\). Then the output of formula \(\varphi_1\) consists of sets \(\{i, j\}\) such that \(S[i]\) is of type \(T\), \(S[j]\) is of type \(R\), \(i < j\), and they satisfy the conditions expressed after the FILTER. By inspecting Figure 1.1, we can see that the pairs satisfying these conditions are \(\{1, 5\}, \{1, 7\}, \{6, 7\}, \{1, 8\}, \{6, 8\}, \ldots\). Formula \(\varphi_1\) illustrates the two most elemental features of CER, namely (non-contiguous) sequencing and local filtering (Cugola & Margara, 2012b; Arasu et al., 2003; Zhang et al., 2014; Abadi et al., 2003; Buchmann & Koldehofe, 2009).

Now imagine we do not want to detect only one reply, but a sequence of replies with the hashtag ‘#iHateTrump’. This can be expressed by using the in the non-contiguous iteration operator \(+\) in the following formula:

\[
\varphi'_1 = (T; R+) \text{FILTER}(T.\text{msg} = \text{‘%#voteTrump%’}) \land R.\text{msg} = \text{‘%#iHateTrump%’}).
\]

In \(\varphi'_1\), the subformula \(R+\) will match the complex events \(\{5, 7, 8\}\), but also all its non-empty subsets \(\{5\}, \{7\}, \{8\}, \{5, 7\}, \ldots\). The motivation for this behaviour is that, like for non-contiguous sequencing, the non-contiguous iteration allows for arbitrary events to be skipped between the events, even if, in this case, they are \(R\) events with the hashtag ‘#iHateTrump’.

Another common feature in CER engines is the disjunction operator \(\text{OR}\) (Cugola & Margara, 2012b; Arasu et al., 2003), which allows to define a pattern that is said to occur whenever at least one of a collection of subpatterns occur. With disjunction, we can express queries like

“A tweet with the hashtag ‘#voteTrump’, followed by either a reply or a tweet with ‘#iHateTrump’.”

This could be expressed by the formula

\[
\varphi''_1 = (T \text{IN}\ FI; (R \text{OR} T) \text{IN}\ SE) \text{FILTER}(FI.\text{msg} = \text{‘%#voteTrump%’}) \land SE.\text{msg} = \text{‘%#iHateTrump%’}).
\]
To understand the meaning of formula $\varphi_1''$, note that $T$ and $R$ refer to the types of events in the stream while $FI$ and $SE$ are variables. Intuitively, we use variables to represent complex events and bind them with the operator $\text{IN}$, to then filter them using predicates over complex events. This way variables $FI$ and $SE$ witness the first event (of type $T$) and the second event (either of type $T$ or $R$), respectively. Note that, in $\varphi_1$, $T$ and $R$ are not assigned to any variable, but they are still used in the filter clause. We use event types themselves also as variables; this generally decreases the number of variables in a formula and aids readability.

1.2. About efficiency in evaluation algorithms

This thesis is heavily focused on efficiency, which is why, before presenting the research questions that motivate this work, we need to talk about the notions of efficiency that we consider. In the last few years, the paradigm of delay-enumeration algorithms has settled as a fair framework to study problems that require the delivery of a potentially massive number of results with respect to the input size (e.g., exponential). They allow to analyse an algorithm and separate the time required to process the input data and compute a (compact) representation of the results, from the time required to deliver the results to the user.

In this work, we will mainly use the notion of output-linear delay enumeration. Consider an input stream $S$ and a query $Q$ that defines a set $Q[S]_i = R_i$ of results at each position $i$, i.e., the set of new results after reading the event at position $i$. In the following, we say that $Q$ is evaluated efficiently if there exists an algorithm $A$ that “processes” $S$ one event at a time, each in time linear over its size, and, after processing every event, is able to enumerate all the results of $R_i$, taking for each one time linear on its size. Later, we will formally define this notion as having linear preprocessing and output-linear delay enumeration.
1.3. Hypothesis and research questions

1.3.1. Efficient enumeration algorithms for CER queries with correlation

Correlation is a key property for a CER language. Recall the formula $\varphi_1$ that looked for USA election debates on Twitter. By judging the given results, one could argue that some of them do not correspond to actual debates. For instance, $\{1,7\}$ is a result but $S[7]$ is not actually a reply to $S[1]$, since $S[7].\text{tweetId}$ differs from $S[1].\text{id}$. By using correlation, one could add this restriction to the pattern, obtaining:

$$\varphi_2 = (T; R) \text{FILTER} (T.\text{msg} = \text{'\#voteTrump'} \land R.\text{msg} = \text{'\#iHateTrump'} \land T.\text{id} = R.\text{tweetId}).$$

This way, only the complex events that satisfy this predicate will be given as results.

Even though the equality predicate $=$ is one of the most important ones, it is clearly not the only possible correlation operator. One might consider, for instance, inequality predicates like $<$ and $\neq$. However, in this work focus on studying only the case of correlation with equalities.

The problem of correlation in query evaluation has been widely studied in theory. However, until now it is not clear how to adopt this results to the setting of CER and, in particular, using the notion of enumeration delay algorithms. This motivates our first question, and its respective hypothesis:

P1: Which fragment of CER queries with correlation can be evaluated efficiently?

H1: There are useful CER queries with correlation that can be evaluated efficiently

1.3.2. Efficient ranked enumeration algorithms

CER engines can retrieve results in an order that is most convenient for the engine to evaluate efficiently, which appears to be an arbitrary order to the end user. However,
for CER users, there are some scenarios where it seems useful to receive the results in a specific order defined by them. This way, the results that are most relevant to the user may be delivered first and, when he recognizes that the remaining results are not of enough utility, he can decide to terminate the enumeration process. This is called a ranked enumeration of the results.

We can find an application of this by considering the fundamental CER operator of windows, whether time or event-based. For instance, in the USA election debate query $\varphi_1$, we might argue that result $\{6, 7\}$ is found to be more relevant than $\{1, 7\}$ at the current time 7, because it represents a more recent debate. Subsequently, we might like to have the capability to prioritize the results that started less time ago. Most CER systems solve this problem by defining an operator called time window, that allows to restrict the stream interval in which the results are searched, whether by giving a time span, called time window (e.g., 10 minutes), or a number of events, called event window (e.g., 100 events). Back to the USA election debate query, if we only want to be alerted about debates within a 10 minute window, we would illustrate it as the following query:

$$
\varphi_3 = (T ; R) \text{ FILTER } (T.\text{msg} = '%\#voteTrump%' \\
\text{\& } R.\text{msg} = '%\#iHateTrump%') \text{ WITHIN } 10 \text{ minutes}.
$$

In this case, operator WITHIN adds to the given results the restriction that the time difference between the first and the last events is no greater than 10 minutes. Then, $\{6, 7\}$ would still be a valid result because the time between events 6 and 7 is lower than 10 minutes. On the other hand, $\{1, 5\}$ and $\{1, 7\}$ would longer be part of the output.

Note that if we had a solution to the ranked enumeration problem, we could use it to evaluate the time window operator. Intuitively, one could define the cost of each result to be the distance between its first and last events. Then, by using the ranked enumeration algorithm to obtain the results with this notion of cost until we reach a
certain cost threshold of $W$, we could effectively obtain all complex events contained within a window of $W$ events.

One might wonder why bother defining special operators for a particular attribute (time), since it could be considered as a case of correlation operator (e.g., by adding in $\varphi_1$ the predicate $R.t_{\text{timestamp}} - T.t_{\text{timestamp}} > 10$ minutes). The main argument is that in CER, time is an attribute with a crucial property: it is always increasing. This can be exploited in the evaluation process to end up with more efficient algorithms.

Windowing has already been studied in the past, and efficient streaming algorithms have been developed. However, to the best of our knowledge, there are no CER algorithms that evaluate event/time windows and provide practical efficiency guarantees.

With the above, we could motivate the question: which fragment of CER queries with event/time windows can be evaluated efficiently? However, as we already stated, this operator can be considered a particular case of ranked enumeration, which is why we consider the following more general question:

P2: Which fragment of CER queries can be evaluated efficiently with ranked enumeration?

H2: There are useful CER queries with ranked enumeration that can be evaluated efficiently

1.4. Contributions

1.4.1. A formal framework

The first contribution (Chapters 2 and 3) is the definition of a CER framework that includes a formal language called Complex Event Logic (CEL) and its regular extension (rCEL), both with a syntax and semantics, an intermediate computational model called Complex Event Automata (CEA) to translate unary rCEL queries into, and an evaluation algorithm to evaluate CEA with constant update time and output-linear delay enumeration.
Our next contribution is related to a special kind of CER operators called selection strategies. Usually in CER systems, selection strategies are applied in a different layer in the query evaluation process: a selection strategy defines a heuristic that is followed during the evaluation of a query to compute only a subset of the complex event results. For instance, the skip-till-next-match strategy in SASE+ defines the heuristic that, every time a new event from the stream is processed, if there is a partial match of the query that can be extended by including this event, then it is extended accordingly. This way, only the complex events that can be composed this way are given in the result to the user. Moreover, usually selection strategies are not defined to provide more expressiveness to the language, but to aid the system in the evaluation process by narrowing the set of results to one more easy to evaluate. This way of treating selection strategies comes with several complications. First, their semantics are often underspecified, leading to unexpected and hard to explain results (or lack of them) in some cases. Secondly, since they are applied in a different layer, they cannot be nested with other operators, disallowing e.g. their application to only fragments of the query and the application of one strategy over another. Therefore, our second contribution is defining selection strategies as in-language operators. We propose a number of selection strategies as regular CEL operators with their own formal semantics. This solves both of the problems stated above. First, since their semantics is user-oriented and not focused on system efficiency, they have a clear and expected behaviour. Second, as they are regular CEL operators, they can be composed with other operators and the resulting query maintains a clear semantics. Further, given this framework, it is not hard to define new selection strategies that might be useful for CER users.

Our third contribution is an expressiveness analysis on fragments of CEL and fragments of rCEL. We define the notion of streaming functions and the so-called ∗-property. A streaming function is every function that maps streams and queries to sets of complex events, like CEL and CEA. Then, a streaming function satisfies the ∗-property if the presence of a complex event in the set of results only depends on the events that constitute such complex event (and, therefore, cannot check any property on the events out of it). Having this notions, we provide several expressiveness results: (1) every CEL formula has the ∗-property; (2) every CEA that has the ∗-property is
expressible by a CEL formula; (3) all operators in the extension rCEL and selection strategies are not expressible in CEL, with the exception of ALL and AND; (4) rCEL with unary predicates captures the exact expressiveness of CEA.

We also provide an comparison between the strict sequencing operators $\cdot$, $\oplus$, and the strict selection strategy $\text{STRICT}$. We show that, when arbitrary predicates are allowed, the strict sequencing operators are strictly more expressive than the strict selection strategy, but they are equally expressive when only unary predicates are allowed.

1.4.2. Partition-By: the CER correlation operator

The first contribution of Chapter 4 is the definition in our CEL framework of the partition-by operator, a common CER operator that is used to define queries with correlation in a restricted manner. Traditionally, this operator partitions the stream into substreams, e.g., by and id value, to then evaluate the query over each of them, effectively returning complex events that have events correlated by their id. We formally define an extension of CEL called pCEL, which includes the partition-by $\text{PART-BY}$. Like for selection strategies, we treat $\text{PART-BY}$ like a normal operator, with a clear semantics that allows it to be composed with other CEL operators.

Our second contribution is the definition of a novel automata model called Chain Complex Event Automata (chain-CEA). As the name suggests, it is inspired by CEA, but modified to support a specific kind of correlation: when processing an event of the stream and building a resulting complex event, the values of the event can be compared with the last event added to the complex event, forming a “chain” of equalities. A number of results are provided for chain-CEA. First, when we only allow equalities, chain-CEA is not closed under I/O-determinization, but this changes if we allow also disequalities, making the model closed under I/O-determinization. Second, every pCEL formula can be translated to an equivalent chain-CEA.

Third, for every chain-CEA there exists a streaming evaluation algorithm with constant update time and output-linear delay enumeration. In practice, we provide an algorithm that receives as input a chain-CEA and processes a stream incrementally,
building a data structure that allows, at any moment, the enumeration of the results at the current time.

Our third contribution is a prerequisite for the result (3) above. We provide an index structure that reads and stores a sequence of tuples over a set $A$ of attribute names and supports an equality-disequality query of the following form. It receives two tuples $t, r$ both having domains that are subset of $A$ and enumerates with constant delay the set of all tuples stored in the structure that are attribute-wise equal with $t$ and attribute-wise different with $r$ (all of the attributes are different than those of $r$).

### 1.4.3. Ranked enumeration

For the following results we abstract beyond the world of CER query evaluation to the more expressive framework of MSO, and present general techniques that allow to evaluate ranked queries. The first contribution of Chapter 5 is the definition of MSO cost functions, a way of defining ranks to MSO assignments over words. Given an MSO structure that encodes a word $w$ and an MSO formula $\varphi$, an MSO function assigns, to every result of the evaluation of $\varphi$ over $w$, a value over an ordered group, therefore defining a (partial) order over the set of assignments. We also define an automata model called cost transducer that outputs assignments and provides a cost for each of them. Further, we provide a construction that, given an MSO formula and a MSO cost function, defines a cost transducer that effectively defines the same set of assignments and keeps costs of the MSO cost function.

The second contribution is an evaluation framework for cost transducer. We provide an algorithm that evaluates a cost transducer over a word with linear preprocessing time and logarithmic delay enumeration. Further, the complexity of the algorithm is inherited directly from the data structure that is used inside the algorithm, meaning that if a data structure with the same interface but an implementation with different complexity is defined, an evaluation algorithm for cost transducer with those complexities would be obtained directly. By working over MSO, this contribution applies directly to different areas and, in particular, to the one of CER.
Our third contribution is a data structure called Heap of Words (HoW), a fully-persistent priority queue designed specially for storing words. This structure stores words with priorities and supports operations for: adding words with priorities, finding/deleting the word with minimum priority, melding two HoW, increasing all priorities by some value, and extending all words with some letter at the end. The last two are non-standard priority queue operations, defined specially for the purpose of the evaluation algorithm. All operations take constant time, except those for finding or deleting the word with minimum priority, the former taking linear time over the length of the word, and the latter taking linear time times a logarithmic factor over the number of operations applied to the structure. It is worth noting that the HoW uses yet another data structure inside it, called Incremental Brodal Queue, an extension we made of the Brodal Queue data structure in order to support a new operation for increasing the priorities of all stored elements.

1.5. Related Work

Next, we present the related work for each field discussed in this thesis.

1.5.1. Formal framework

CER systems are usually divided into three approaches (Cugola & Margara, 2012b; Giatrakos et al., 2020; Alevizos et al., 2017; Artikis, Margara, Ugarte, Vansummeren, & Weidlich, 2017): automata-based, tree-based, and logic-based, with some systems, e.g., (Esper Enterprise Edition website, n.d.; Cugola & Margara, 2010), being hybrids; we refer the reader to the surveys (Cugola & Margara, 2012b; Giatrakos et al., 2020; Alevizos et al., 2017; Artikis et al., 2017) for in-depth discussion of each class of systems. Automata-based systems are close to what we propose in this work. They typically propose a CER query language that is inspired by regular expressions, which is evaluated by means of custom automata models. Previous proposals, e.g., SASE (Agrawal, Diao, Gyllstrom, & Immerman, 2008), NextCEP (Schultz-Møller, Migliavacca, & Pietzuch, 2009), DistCED (Pietzuch, Shand, & Bacon, 2003), do not provide denotational semantics for their language; the output of queries is defined by intermediate automata models. This implies that either iteration cannot be nested
(Agrawal et al., 2008) or its semantics is confusing (Schultz-Møller et al., 2009). Other proposals, e.g., CEDR (Barga, Goldstein, Ali, & Hong, 2007), TESLA (Cugola & Margara, 2010), PBCED (Akdere, Çetintemel, & Tatbul, 2008), have formal semantics, but they do not include iteration. An exception is Cayuga (Demers, Gehrke, Hong, Riedewald, & White, 2005), but their sequencing operator is non-associative, which results in unintuitive semantics. Our framework is comparable to these systems, but provides a well-defined formal semantics that is compositional, allowing arbitrary nesting of operators.

Extensions of regular expressions with data filtering capabilities have been considered outside of the CER context. *Extended regular expressions* (Aho, 1990; Câmpeanu, Salomaa, & Yu, 2003; Carle & Narendran, 2009) extend the classical regular expressions operating on strings with variable binding expressions and variable backreference expressions. Variables binding expressions can occur inside a Kleene closure but, when being referred, a variable always refers to the last binding. Extended regular expressions differ from CEL in that they operate on finite strings over a finite alphabet rather than infinite streams over an infinite alphabet of possible events and, further, they use variables only to filter the input and not to construct the output. Regular expressions with variable bindings have also been considered in the so-called spanners approach to information extraction (Fagin, Kimelfeld, Reiss, & Vansummeren, 2015). There, however, variables are only used to construct the output and cannot be used to inspect the input. In addition, variable binding inside Kleene closures is prohibited.

Finally, there has been some research in theoretical aspects of CER, e.g., in axiomatization of temporal models (White, Riedewald, Gehrke, & Demers, 2007), privacy (He, Barman, Wang, & Naughton, 2011), and load shedding (He, Barman, & Naughton, 2014). This literature does not study the semantics and evaluation of CER and, therefore, is orthogonal to our work.

1.5.2. Partition-By

New techniques in dynamic query evaluation (Ceri & Widom, 1991; Ahmad, Kennedy, Koch, & Nikolic, 2012; Chirkova, Yang, et al., 2012) have recently attracted
a lot of attention (Berkholz, Keppeler, & Schweikardt, 2017; Idris, Ugarte, & Vansummeren, 2017; Idris, Ugarte, Vansummeren, Voigt, & Lehner, 2018). In (Berkholz et al., 2017; Idris et al., 2017), the streaming evaluation of CQ is considered but this does not include queries with order. In (Idris et al., 2018), inequalities over atoms are considered, but only for the case of CQ. Our setting also includes disjunction and iteration (but not conjunction), which makes our work orthogonal to the work in (Berkholz et al., 2017; Idris et al., 2017, 2018).

In recent years, automata models have been proposed that, while not yet applied to the CER domain, allow a form of event correlation. In particular, register automata (“Finite-memory automata”, 1994), data automata (Bojańczyk, David, Muscholl, Schwentick, & Segoufin, 2011), and class memory automata (“On notions of regularity for data languages”, 2010) operate on so-called data words: strings in which each symbol has an associated data value drawn from an infinite set $D$. These automata have finite state control, but allow mechanisms to compare data values occurring at different positions in the data word. This corresponds to event correlation in the CER domain. It is known, however, that register automata are limited in expressive power, while the data automata (Bojańczyk et al., 2011) and class memory automata (“On notions of regularity for data languages”, 2010), are not closed under Kleene closure (i.e., iteration). As such, it is not a priori clear how they apply to the CER setting if we insist that iteration is compositional. Recently, in (Alevizos, Artikis, & Paliouras, 2018) a similar extension of complex event automata with registers was proposed. However, this work does not study the determinization and evaluation of this model with constant update time and bounded delay enumeration.

1.5.3. Ranked enumeration

Several people have studied the enumeration problem over words following different formalisms. For example, (Bagan, 2006; Courcelle, 2009; Segoufin, 2013) studied the enumeration problem for MSO logic, and (Florenzano, Riveros, Ugarte, Vansummeren, & Vrgoc, 2020; Amarilli, Bourhis, Mengel, & Niewerth, 2019) for regular
spanners. For all these formalisms, it is shown that there exists an enumeration algorithm with linear time preprocessing and constant delay, both in data complexity (i.e., in the size of the input word).

The approach of “scoring” solutions to quickly provide relevant answers to the user has been used particularly in the context of information extraction. Indeed, there have been several recent proposals (Doleschal, Kimelfeld, Martens, & Peterfreund, 2020; Doleschal, Bratman, Kimelfeld, & Martens, 2021) to extend document spanners with annotations from a semi-ring. The proposed annotations are typically useful to capture the confidence of each solution (Doleschal et al., 2020). For instance, (Doleschal et al., 2021) proves that the enumeration of the answers following their scores’ order is possible with polynomial-time preprocessing and polynomial delay.

In this thesis we define data structures that are fully-persistent (Driscoll, Sarnak, Sleator, & Tarjan, 1989), i.e., that each operation applied to the structure returns a new object without changing the previous one. To obtain the required efficiency, we rely on a classical persistent data structure called Brodal queue that we extend in order to implement our required interface.

For ranked query evaluation, there has been recent progress in the context of conjunctive queries: on the efficient computation of top-$k$ queries (Tziavelis, Gatterbauer, & Riedewald, 2020) and the efficient ranked enumeration (Tziavelis, Ajwani, Gatterbauer, Riedewald, & Yang, 2020; Deep & Koutris, 2019). These advances consider relational data (which is more general than words) and conjunctive queries (which is more restricted than MSO queries); they are thus incomparable to our work.

1.6. Outline

In Chapter 2, we provide the common notions that will be used throughout the rest of the work: definition of the basic objects, syntax and semantics of the common core logic, and the notions of efficiency that we consider.

In Chapter 3, we provide the full-fledged regular framework for CER. Section 3.1 provides rCEL, the extended regular query language for CER; Section 3.2 defines CEA, the computational model for evaluating rCEL; Section 3.3 compares the unary
fragment of rCEL with CEA in terms of their expressiveness; Section 3.4 formalizes the notion of selection strategies in the framework; Section 3.5 studies the relationship in terms of expressiveness between contiguous operators and contiguous selection strategies; finally, Section 3.6 shows how to efficiently evaluate CEA and, therefore, rCEL.

In Chapter 4, we extend CEL with correlation capabilities, in the form of a new operator called partition-by. Section 4.1 formalizes the partition-by operator; Section 4.2 proposes chain-CEA an extension of CEA able to express CEL queries with partition-by; Section 4.3 shows how we can efficiently evaluate chain-CEA; finally, Section 4.4 presents in detail the implementation of index structure used in the evaluation that allows to handle equalities and disequalities.

In Chapter 5, we abstract beyond the world of CER query evaluation to the more expressive framework of MSO. Section 5.1 introduces the ranked enumeration problem for MSO queries on words; Section 5.2 presents MSO cost functions and state the main result: an efficient evaluation algorithm for the ranked enumeration problem; Sections 5.3.1 and 5.3.2 show applications of the main result to the settings of CER and document spanners, respectively; Section 5.4 describes our enumeration scheme that rely on two data structures: the Heap of Words described in Section 5.5 and the incremental Brodal queues presented in Section 5.6.

1.7. Previous Publications

This work is a compound of four conference papers and a journal paper. Chapters 2 and 3 are based on the journal paper “A formal framework for complex event recognition” (Grez, Riveros, Ugarte, & Vansummeren, 2021), which at the same time is a compound of the two conference papers, “A formal framework for complex event processing” (Grez, Riveros, & Ugarte, 2019) and “On the expressiveness of languages for complex event recognition” (Grez, Riveros, Ugarte, & Vansummeren, 2020). Chapters 4 and 5 are based on the conference papers “Towards streaming evaluation of queries with correlation in complex event processing” (Grez & Riveros, 2020) and “Ranked enumeration of mso logic on words”(Bourhis, Grez, Jachiet, & Riveros, 2020).
2020), respectively. All of the above have been revised and modified for this work in order to present them with a unified notation and provide a wider view of the results and the relations between them. Also, in this work all results are provided with complete proofs, which in most cases were missing in their paper versions.
Chapter 2. PRELIMINARIES

In this section we provide some basic notation that will be common through the whole document. We formally introduce Complex Event Logic (CEL for short) with its core operators. Finally, we present the computational model that our results are stated on, and the computational assumptions that we consider throughout our work.

2.1. Schemas, tuples and streams

Let $A$ be a set of attribute names and $D$ a set of values. A database schema $\mathcal{R}$ is a finite set of relation names, where each $R \in \mathcal{R}$ is associated to a finite set of attributes $\text{att}(R) \subseteq A$. If $R$ is a relation name, then an $R$-tuple is a function $t : \text{att}(R) \to D$; as notation, we say that $\text{type}(t) = R$. For any relation name $R$, $\text{tuples}(R)$ denotes the set of all possible $R$-tuples, i.e., $\text{tuples}(R) = \{ t : \text{att}(R) \to D \}$. Similarly, for any database schema $\mathcal{R}$, $\text{tuples}(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} \text{tuples}(R)$. Throughout the rest of this work, we fix a schema $\mathcal{R}$.

A stream is an infinite sequence $S = t_0 t_1 \ldots$ where $t_i \in \text{tuples}(\mathcal{R})$. Given a stream $S = t_0 t_1 \ldots$ and a position $i \in \mathbb{N}$, the $i$-th element of $S$ is denoted by $S[i] = t_i$, and the sub-stream $t_i t_{i+1} \ldots$ of $S$ is denoted by $S_i$. Note that we consider that the time of each event is given by its index, and defer a more elaborated time model (like (White et al., 2007)) to future work.

Let $L$ be a finite set of variables. We assume that $L$ contains all relation names (i.e., $\mathcal{R} \subseteq L$). A CEL predicate of arity $n$ is an $n$-ary relation $P$ over sets of tuples, i.e., $P \subseteq (2^{\text{tuples}(\mathcal{R})})^n$. We write $\text{arity}(P)$ for the arity of $P$. Let $\mathcal{P}$ be a set of CEL predicates. An atom over $\mathcal{P}$ is an expression of the form $P(A_1, \ldots, A_n)$ where $P \in \mathcal{P}$ is a predicate of arity $n$, and $A_1, \ldots, A_n \in L$. We also write $P(\bar{A})$ for $P(A_1, \ldots, A_n)$ when convenient.

\footnote{In our framework, if two relation names $R_1$ and $R_2$ have the same set of attributes, then every $R_1$-tuple is also an $R_2$-tuple, and vice-versa. In other words, $R_1$ and $R_2$ will be indistinguishable. If this is undesirable, one can always extend the schema and add the original relation name as a special attribute.}
\[ \varphi ::= R \quad \text{R-tuple selection} \]

\[ | \varphi \text{ IN } A \quad \text{Variable binding} \]

\[ | \varphi \text{ FILTER } P(\vec{A}) \quad \text{Filtering (both local & correlation)} \]

\[ | \varphi ; \varphi \quad \text{Non-contiguous sequencing} \]

\[ | \varphi + \quad \text{Non-contiguous iteration} \]

\[ | \varphi \text{ OR } \varphi \quad \text{Disjunction} \]

Figure 2.1. Syntax of CEL. \( R \) ranges over relation names, \( A \) over variables in \( L \), \( P(\vec{A}) \) over atoms over \( P \), and \( L \) over subsets of \( L \).

2.2. CEL syntax and semantics

The syntax of CEL is given by the grammar in Figure 2.1. There, \( R \) ranges over relation names, \( A \) over variables in \( L \), \( P(\vec{A}) \) over atoms over \( P \), and \( L \) over subsets of \( L \). Unlike existing frameworks, we do not restrict the syntax and allow arbitrary nesting (in particular of the iteration operators \( + \) and \( \oplus \)).

A notable feature of CEL is that variables bind to complex events. By contrast, some existing CER languages bind variables to atomic events (i.e., individual tuples). While it is possible to introduce a variant of CEL that binds atomic events, we find that the current version, with binding to complex events instead, simplifies the definition of both the syntax and semantics. In particular, binding to atomic events significantly complicates the semantics of iteration. See (Grez et al., 2020) for an in-depth discussion.

To formally define the semantics of CEL we first need to introduce some auxiliary concepts and notation. A complex event \( C \) is defined as a (possibly empty) finite subset of \( \mathbb{N} \). Intuitively, a complex event contains the positions of the events that witness the matching of a formula over a stream. We denote by \( |C| \) the size of \( C \) and, if \( C \) is non-empty, by \( \min(C) \) and \( \max(C) \) the minimum and maximum element of \( C \), respectively. Given a stream \( S \) and complex event \( C \) we define \( S[C] = \{ S[i] \mid i \in C \} \) to be the set of tuples in \( S \) positioned at the indices specified by \( C \).

Valuations are the formal constructs by which variables bind complex events in CEL. Formally, a valuation is a function \( \mu : L \rightarrow 2^{\mathbb{N}} \) that maps variables to complex events. The support of \( \mu \) is the set of all positions appearing in complex events in
the range of $\mu$, $\sup(\mu) = \bigcup_{A \in \mathbb{L}} \mu(A)$. We denote by $A \mapsto C$ the valuation $\mu$ such that $\mu(A) = C$ and $\mu(B) = \emptyset$ for $B \in \mathbb{L} \setminus \{A\}$. For a valuation $\mu$ we denote by $\mu|_A \mapsto C$ the valuation $\mu'$ that equals $\mu$ on all variables except $A$, which it maps to $C$. Furthermore, if $L \subseteq \mathbb{L}$ we denote by $\mu|_L$ the restriction of $\mu$ to $L$: this is the valuation $\mu'(A) = \mu(A)$ for every $A \in L$ while $\mu'(A) = \emptyset$ when $A \notin L$. We define the union between two valuations $\mu_1$ and $\mu_2$ by $(\mu_1 \cup \mu_2)(A) = \mu_1(A) \cup \mu_2(A)$ for every $A \in \mathbb{L}$.

To define the semantics of a CEL formula $\varphi$, it is convenient to first define an auxiliary semantics $\llbracket \varphi \rrbracket$ that returns valuations, and to then define the (final) semantics $\llbracket \varphi \rrbracket$ that returns complex events based on this auxiliary semantics. Concretely, the auxiliary valuation semantics of $\varphi$ over a stream $S$, starting at position $i$ and ending at position $j \geq i$, denoted $\llbracket \varphi \rrbracket(S, i, j)$, is defined by induction on the structure of $\varphi$ as shown in Figure 2.2.

The (final) semantics of a CEL formula $\varphi$ on $S$ starting at $i$ and ending at $j$ is then defined as

$$\llbracket \varphi \rrbracket(S, i, j) = \{\sup(\mu) \mid \mu \in \llbracket \varphi \rrbracket(S, i, j)\}.$$ 

In other words, the complex event semantics is given by the valuation semantics by collecting all events in the range of the valuations, thereby essentially “forgetting” the variables that were introduced during the evaluation of $\varphi$.

We will often be interested in the result of evaluating a formula from the start of the stream. For every $n \in \mathbb{N}$ we abbreviate $\llbracket \varphi \rrbracket(S, 0, n)$ by $\llbracket \varphi \rrbracket_n(S)$, and similarly $\llbracket \varphi \rrbracket(S, 0, n)$ by $\llbracket \varphi \rrbracket_n(S)$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure22.png}
\caption{CEL semantics.}
\end{figure}
2.2.1. Notational conventions

Throughout this work we use $\varphi \text{ FILTER } (P(\bar{A}) \land Q(\bar{B}))$ or $\varphi \text{ FILTER } (P(\bar{A}) \lor Q(\bar{B}))$ as syntactic sugar for $(\varphi \text{ FILTER } P(\bar{A})) \text{ FILTER } Q(\bar{B})$ and $(\varphi \text{ FILTER } P(\bar{A})) \text{ OR } (\varphi \text{ FILTER } Q(\bar{B}))$, respectively. Furthermore, if $A$ is a variable, $x$ is an attribute name, and $c$ is a constant, then we write $A.x = c$ for the atom $P(A)$ with predicate $P$ defined as $P = \{ A \subseteq \text{tuples}(\mathcal{R}) \mid \forall t \in A: t(x) = c \}$. We use similar notation to denote inequality comparison (e.g., $A.x < 10$). By an expression like $H.id = T.id$ we denote the atom $P(H, T)$ where $P$ is the binary predicate $\{(H, T) \in 2^{\text{tuples}(\mathcal{R})} \times 2^{\text{tuples}(\mathcal{R})} \mid \forall h \in H, \forall t \in T, t(id) = h(id)\}$.

We use the following notation to refer to extension of CEL. Given a set of operators $O$, let $\text{CEL} \cup O$ be the fragment of CEL formulas that can be defined by using operators of CEL (i.e., IN, FILTER, ;, +, OR) plus the operators in $O$.

2.2.2. Unary CEL

We will restrict our study of efficient evaluation of CEL formulas to those formulas in CEL that can be considered “regular”. In particular, such formulas can compare events only on their arrival order, not on their attribute values (except for some cases, most notably the partition-by operator of Chapter 4).

We will refer to the regular formulas of CEL (and its extensions) as unary formulas, whose formal definition is as follows. A first order predicate (FO predicate for short) is any decidable subset of $\text{tuples}(\mathcal{R})$. FO predicates hence distinguish themselves from CEL predicates in that the former test properties of individual $\mathcal{R}$-tuples, whereas the latter test entire complex events (i.e., sets of $\mathcal{R}$-tuples), or tuples of complex events. Throughout this work we assume given a fixed universe $\mathcal{U}$ of FO predicates, which contains at least the empty predicate $\emptyset$; and for every relation symbol $R \in \mathcal{R}$, the predicate $\text{tuples}(R)$; and which is closed under union, intersection and difference (i.e., if $P_1, P_2 \in \mathcal{U}$, then so are $P_1 \cup P_2$, $P_1 \cap P_2$ and $P_1 \setminus P_2$).

If $P$ is a FO predicate, then we denote by $P^{SO}$ its second order extension, which is the CEL predicate such that $P^{SO} = \{ S \subseteq \text{tuples}(\mathcal{R}) \mid P(t) \text{ for every } t \in S \}$. We
extend this definition to sets of FO predicates: if $\mathcal{P}$ is a set of FO predicates, then $\mathcal{P}^{SO}$ is the set \( \{ P^{SO} \mid P \in \mathcal{P} \} \).

**Definition 1.** A CEL formula $\varphi$ is unary if for every subformula of the form $\text{FILTER } P(\bar{A})$, it holds that $P(\bar{A})$ is the second order extension of a FO predicate (i.e. $P(\bar{A}) \in U^{SO}$).

We remark that neither correlation nor windowing are expressible in unary CEL in general.

### 2.3. Efficiency in Complex Event Recognition

We next specify what sort of algorithms we allow to solve the upcoming problems, and what the efficiency guarantees are that we would like to achieve. For the algorithms, we assume the model of Random Access Machines (RAM) with uniform cost measure, and addition and subtraction as basic operations (Aho & Hopcroft, 1974). This implies, for example, that the access to a lookup table (i.e., a table indexed by a key) takes constant time. These are common assumptions in the literature of enumeration algorithms (Durand & Grandjean, 2007; Segoufin, 2013). Further, we assume that the stream $S$ can be read by calling a special instruction $\text{yield } S$ that returns the next unprocessed event of $S$ and places it in the RAM's read-only input registers. To process each event, the machine has read-write work registers where it does the computation, and write-only output registers where it enumerates the complex events.

Defining a notion of efficiency for the evaluation is challenging since we would like to compute complex events in one pass over $S$ while using a restricted amount of resources. Streaming algorithms (Ikonomovska & Zelke, 2013; Golab & Özsu, 2003) are a natural starting point for a notion of efficiency. These algorithms usually restrict the time allowed to process each tuple and the space needed to process the first $n$ items of a stream (e.g., constant or logarithmic in $n$). However, an important difference with CER is that the arrival of a single event might generate an exponential number of complex events as output. To overcome this problem, we propose to divide the evaluation in two: (1) consuming new events and updating the system’s internal
memory, and (2) generating complex events from the system’s internal memory. We require both parts to be as efficient as possible: (1) should process each event in a time that does not depend on the number of events seen in the past and (2) should not spend any time processing. Instead, it should be completely devoted to generating the output.

Our notion of efficiency for CER is therefore formalized as follows. Let $\mathcal{U}$ be the set of unary predicates. We restrict $\mathcal{U}$ to contain unary predicates with constant time evaluation, namely, for every predicate $P$ in $\mathcal{U}$ and every tuple $t$, we assume that checking whether $t \in P$ takes constant-time. Furthermore, if $t$ is a tuple, then let $|t|$ denote the number of RAM registers required to encode $t$. We will assume that every tuple fits in a single register, namely that $|t| = 1$. For a function $f : \mathbb{N} \to \mathbb{N}$, a CER evaluation algorithm with $f$-update time and $g$-delay is an algorithm that evaluates a formula $\varphi$ of CEL (or an extension of CEL) over a stream $S = t_0 t_1 \ldots$. It receives as input $\varphi$ and reads $S$ by calling the $\text{yield}_S$ method sequentially. For every $n \geq 0$, after the $n$-th call the evaluation algorithm processes the event $t_n$ in two phases.

(I) In the first phase, called the update phase, the algorithm updates a data structure $D$ with $t_n$ and the time spent is bounded by $O(f(|\varphi|))$.

(II) The second phase, called the enumeration phase, occurs immediately after the first phase and outputs $[\varphi]_n(S)$ by using $D$. More specifically, during this phase the algorithm: (1) writes $\#C_1\#C_2\#\ldots\#C_m\#$ to the output registers where $\#$ is a distinct separator symbol, and $C_1, \ldots, C_m$ is an enumeration (without repetitions) of all complex events in the set $[\varphi]_n(S)$, (2) it writes the first $\#$ as soon as the enumeration phase starts, and (3) it stops immediately after writing the last $\#$.

The purpose of separating the processing of each event in two phases is to be able to distinguish between the time required to update the data structure and the time to enumerate all complex events. Moreover, this distinction allows to measure the delay between two outputs as follows. Let $m_n$ denote the number of $\#'$s written during the enumeration phase of of the $n$-th event on $S$. Let $\text{time}_i(\varphi, S, n)$ denote the time, during the enumeration phase of the $n$-th event of $S$, that the algorithm writes the $i$-th $\#$, for $1 \leq i \leq m_n$. Define $\text{delay}_i(\varphi, S, n) = \text{time}_{i+1}(\varphi, S, n) - \text{time}_i(\varphi, S, n)$ for
$1 \leq i \leq m_n - 1$. We say that the algorithm has \textit{g-delay} if, for all stream positions $n$, (1) if $\llbracket \varphi \rrbracket_n(S)$ is non-empty then $\text{delay}_i(\varphi, S, n) \in \mathcal{O}(g(n) \cdot |C_i|)$ for every $i \leq |\llbracket \varphi \rrbracket_n(S)|$; and (2) if $|\llbracket \varphi \rrbracket_n(S)| = 0$ then $\text{delay}_i(\varphi, S, 1) = \mathcal{O}(g(n))$. In other words, the time between writing the $i$-th # and $(i + 1)$-th # is at most the size of the $i$-th output $C_i$ of $\llbracket \varphi \rrbracket_n(S)$ times $g(n)$, and is $g(n)$ if $\llbracket \varphi \rrbracket_n(S)$ is empty, up to a constant factor. A particular case that will be recurrent in our work is when the function $g$ is constant, i.e., $g \in \mathcal{O}(1)$. In that case, we say that the algorithm has \textit{output-linear delay}.

Concerning (I), note that in practice the formula $\varphi$ is generally small, especially compared to the unbounded length of the stream. Viewed from the perspective of data complexity (Abiteboul, Hull, & Vianu, 1995) where the size of the query is assumed to be fixed, an update time of $\mathcal{O}(f(\mid \varphi \mid))$ then amounts to constant update time. Then, under the assumptions that we make of having a constant schema, constant query size $\mid \varphi \mid$, and all data values fitting in a single register, (I) implies that the update phase takes constant time update per tuple.

In Chapter 5 we use the alternative notion of preprocessing, which is useful in the scenario where the input, instead of being an infinite stream, is a finite word. This notion is presented formally in Section 5.1.
Chapter 3. A FORMAL FRAMEWORK

In this chapter we give a more in-depth analysis of the previously presented CEL logic and its framework. In the following we will cover the topics of language design, query compilation and efficient evaluation.

First, we provide an extension of CEL that contains additional operators that, while not being considered basic and not being present in most CER languages, should still be taken into account in order to provide a thorough analysis of the framework. These new operators allow to express notions like negation, projection and contiguousness among events, while still maintaining the compositional syntax and clear semantics of the logic. As we will see later, some of these operators end up providing additional expressiveness to the logic; others, on the other hand, do not, and thus should be included in the systems only as syntactic sugar.

We formally define selection strategies as individual CER operators in their own right, and study in detail the difference in expressive power between non-contiguous operators, contiguous operators, and selection strategy. As expected, contiguous operators are found more expressive than non-contiguous ones. More interestingly, we show that contiguous operators are more expressive than the popular STRICT selection strategy (often called strict-contiguity in the literature) in general, but that contiguous operators have the same expressive power as STRICT when event correlation (the ability to compare events on other attributes than their arrival order) is not allowed.

With a well-defined CER language at hand, we turn our attention to query evaluation. Many CER systems use automata-based models for query evaluation, either exclusively (Demers et al., 2006; Agrawal et al., 2008; Zhang et al., 2014; Pietzuch et al., 2003; Schultz-Møller et al., 2009; Apache FlinkCEP, n.d.) or in combination with dataflow-like operators (Wu et al., 2006) or other search strategies (Cugola & Margara, 2010; Akdere et al., 2008). Unfortunately, however, these automata models are complicated (Pietzuch et al., 2003; Schultz-Møller et al., 2009), informally defined (Demers et al., 2006; Akdere et al., 2008) or non-standard (Cugola & Margara, 2010; Agrawal et al., 2008; Zhang et al., 2014). In practice, this implies that, although
finite state automata are a recurring approach in CER, there is no general evaluation strategy with clear performance guarantees. This is true even for queries without event correlation and aggregation. Such queries intuitively form the “regular” fragment of CER queries, and should therefore be an ideal target for automaton-based techniques.

Then, we introduce a formal, automaton-based computational model for unary CEL, called Complex Event Automata (CEA). Complex event automata are a form of finite state automata that draw upon the rich and established literature of formal languages and automata theory, most notably symbolic automata (Veanes, 2013) and finite state transducers (Berstel, 2013), to be applicable to the CER domain. While readers familiar with formal languages and automata theory may find the definition of CEA standard and straightforward, we believe that this is CEA’s strength as a candidate “standard” evaluation model for the regular core of CER.

We study the properties of CEA and its relationship to unary CEL. Concretely, we show that CEA are closed under so-called I/O-determinization and provide translations for unary CEL formulas into CEA, and vice versa. Further, we demonstrate that all selection strategies within the scope of this work can be compiled into CEA. Finally, we identify a fragment of CEA that coincides with unary CEL when one can express only non-contiguous sequencing and iteration.

These results unify the evaluation process of unary CEL and selection strategies into one problem: the evaluation of the CEA model. As the main result of this chapter, we then describe an algorithm for evaluating CEA with strong performance guarantees: constant time to process each tuple of the stream (under certain assumptions) followed by output-linear delay enumeration of the output. This complexity is optimal since any evaluation algorithm needs to at least inspect every input tuple and generate the query answers. We stress that, in particular, the runtime of our algorithm is independent of the number of partial matches, i.e., the number of partial complex events that may be completed and output in the future if suitable events occur later in the stream. Depending on the query and the stream, the number of partial matches may be significantly bigger than the number of complete complex events that are output. In particular, it has been repeatedly observed (e.g., (Giatrakos et al., 2020; Zhang et al., 2014)) that in
\( \varphi ::= \varphi : \varphi \)

- **Contiguous sequencing**

| \( \varphi \oplus \) |
| \( \pi_L(\varphi) \) |
| \( \text{START}(\varphi) \) |
| \( \varphi \text{ AND } \varphi \) |
| \( \varphi \text{ UNLESS } \varphi \) |

- **Contiguous iteration**

- **Variable projection**

- **Anchoring**

- **Conjunction**

- **Interleaved conjunction**

- **Guarded negation**

**Figure 3.1.** Additional syntax rules of rCEL. \( L \) ranges over subsets of \( L \).

\[
\begin{align*}
\llbracket \varphi_1 : \varphi_2 \rrbracket(S, i, j) &= \{ \mu_1 \cup \mu_2 \mid \exists k, i \leq k < j : \\
&\quad \mu_1 \in \llbracket \varphi_1 \rrbracket(S, i, k), \mu_2 \in \llbracket \varphi_2 \rrbracket(S, k + 1, j), \\
&\quad \max(\sup(\mu_1)) = k, \min(\sup(\mu_2)) = k + 1 \}, \\
\llbracket \varphi_1 \oplus \varphi_2 \rrbracket(S, i, j) &= \llbracket \varphi_1 \rrbracket(S, i, j) \cup \llbracket \varphi_2 \rrbracket(S, i, j) \\
\llbracket \pi_L(\varphi) \rrbracket(S, i, j) &= \{ \mu \mid L(\mu) \in \llbracket \varphi \rrbracket(S, i, j) \} \\
\llbracket \text{START}(\varphi) \rrbracket(S, i, j) &= \{ \mu \mid \text{min}(\sup(\mu)) = i \} \\
\llbracket \varphi_1 \text{ AND } \varphi_2 \rrbracket(S, i, j) &= \llbracket \varphi_1 \rrbracket(S, i, j) \cap \llbracket \varphi_2 \rrbracket(S, i, j) \\
\llbracket \varphi_1 \text{ ALL } \varphi_2 \rrbracket(S, i, j) &= \{ \mu_1 \cup \mu_2 \mid \exists i_1 \leq j_1, i_2 \leq j_2 : \\
&\quad \mu_1 \in \llbracket \varphi_1 \rrbracket(S, i_1, j_1), \mu_2 \in \llbracket \varphi_2 \rrbracket(S, i_2, j_2), \\
&\quad i = \min\{i_1, i_2\}, j = \max\{j_1, j_2\}, \} \\
\llbracket \varphi_1 \text{ UNLESS } \varphi_2 \rrbracket(S, i, j) &= \{ \mu \in \llbracket \varphi_1 \rrbracket(S, i, j) \mid \llbracket \varphi_2 \rrbracket(S, i', j') = \emptyset \text{ for all } i \leq i' \leq j' \leq j \}
\end{align*}
\]

**Figure 3.2.** rCEL semantics.

the presence of non-contiguous sequencing and iteration, maintaining partial matches becomes a bottleneck and is only practically feasible when one restricts detection to short time windows. In contrast, our evaluation algorithm is not prone to this behavior.

### 3.1. A Query Language for CER

In this section we present the full-fledged regular extension of CEL, which we denote with the name rCEL. More specifically, rCEL corresponds to the extension \( \text{CEL} \cup \{ : , \oplus, \pi, \text{START, AND, ALL, UNLESS} \} \), with the additional operations and their syntax are shown in Figure 3.1. Unlike existing frameworks, we do not restrict the syntax and allow arbitrary nesting (in particular of the iteration operators + and \( \oplus \)).
Like for CEL, to define the semantics of a rCEL formula $\varphi$, we first define an auxiliary semantics $\llbracket \varphi \rrbracket$ that returns valuations. The auxiliary valuation semantics of $\varphi$ over a stream $S$, starting at position $i$ and ending at position $j \geq i$, denoted $\llbracket \varphi \rrbracket(S, i, j)$, is defined by induction on the structure of $\varphi$ as shown in Figure. 3.2. The (final) semantics of a rCEL formula $\varphi$ on $S$ starting at $i$ and ending at $j$ is then defined the same as for CEL formulas: $\llbracket \varphi \rrbracket(S, i, j) = \{ \text{sup}(\mu) \mid \mu \in \llbracket \varphi \rrbracket(S, i, j) \}$. Also, we maintain for rCEL the abbreviations $\llbracket \varphi \rrbracket_n(S)$ and $\llbracket \varphi \rrbracket_n(S)$ that refer to $\llbracket \varphi \rrbracket(S, 0, n)$ and $\llbracket \varphi \rrbracket(S, 0, n)$, respectively. Finally, we extend the notion of unary formulas to rCEL as expected.

**Definition 2.** A rCEL formula $\varphi$ is unary if for every subformula of the form $\varphi' \FILTER P(\overline{A})$, it holds that $P(\overline{A})$ is the second order extension of a FO predicate (i.e. $P(\overline{A}) \in U^{SO}$).

We note that the contiguous sequencing operator in a formula $\varphi_1 : \varphi_2$ only requires the maximal position in the complex event output by $\varphi_1$ to be consecutive with the minimal position in the complex event output by $\varphi_2$. Because $\oplus$ is defined through repeated application of $\oplus$, this implies that a formula $\varphi \oplus$ does not impose that all events matched by $\varphi$ in single iteration appear contiguous. For example, the formula $(R; S)\oplus$ imposes that the last event $S$ of one iteration occurs right before the first event $R$ of the next iteration, but in one iteration the $R$ event and the $S$ event do not need to occur contiguously. By contrast, the formula $(R : S)\oplus$ does require $R$ and $S$ to be contiguous.

### 3.1.1. Discussion of language features.

Table 3.1 lists a set of CER operators that, according to recent surveys in the field (Giatrakos et al., 2020; Alevizos et al., 2017), can be considered “basic operators that should be present in every CER language” (Giatrakos et al., 2020). If we relate rCEL to the operators mentioned in this table, then it is clear that rCEL includes sequencing (both contiguous and non-contiguous), disjunction, iteration (contiguous and non-contiguous), conjunction (two forms), negation, and filtering (with arbitrary predicates, hence supporting both local filtering and correlation). If we restrict to unary
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>rCEL</th>
<th>rCEL</th>
<th>CEA</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequencing</strong></td>
<td>Two patterns following each other temporally</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Disjunction</strong></td>
<td>Either of two patterns occurring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Iteration</strong></td>
<td>A pattern occurring repeatedly</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Local filters</strong></td>
<td>Filler complex events based on local properties of the constituent events</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Correlation</strong></td>
<td>Filter complex events based on relations among the constituent events</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><strong>Conjunction</strong></td>
<td>Matching multiple patterns at the same time, regardless of the temporal relation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Negation</strong></td>
<td>Absense of a pattern</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><strong>Projection</strong></td>
<td>Transforming attribute values of simple events</td>
<td>✗</td>
<td>✗</td>
<td>✓ a</td>
<td></td>
</tr>
<tr>
<td><strong>Windowing</strong></td>
<td>Limit complex events to a specified time window</td>
<td>✓</td>
<td>✗</td>
<td>✓ b</td>
<td></td>
</tr>
<tr>
<td><strong>Selection strategies</strong></td>
<td>Control if and how “irrelevant” events may occur</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 3.1. Basic CER operators, and how they are included in the languages considered in this paper. (a) rCEL does not have an operator that transforms the set of attributes present in simple events. Note, however, that it does have a projection operator that projects on tuples of complex events. (b) While rCEL does not have a dedicated windowing operator, windowing is expressible by means of filtering (although non-unary filtering is required). See Chapter 5 (Section 5.3.1) for a formal definition of a dedicated windowing operator.

rCEL, then only correlation is no longer supported, although in Chapter 4 we present the partition-by operator which allows for a restricted type of correlation while still using only unary predicates in general.

The remaining operators of Table 3.1 require more discussion.

While rCEL does have a projection operator, this operator projects *valuations*. By contrast, the projection operator referred to in Table 3.1 is meant to project away attributes occurring in primitive events, mostly for the purpose of displaying matched complex events to a user. For example, using the projection operator of Table 3.1, one could specify that only the `id` attribute of the stream in Fig. 1.1 should be returned. While this feature may be important in a practical language, it is non-essential from the viewpoint of recognizing complex events, and we therefore do not consider this
operator further. If desired, such projection can always be done after recognition is complete.

While rCEL does not have an explicit windowing operator, windowing is expressible using filtering. For example, assume that we wish to evaluate formula $\varphi_1$ over a sliding window consisting of 100 events. Assume that every relation $R \in \mathcal{R}$ has an attribute $ts$ that records the position of each $R$-tuple in the stream. Further assume that $Q$ is the unary predicate that checks that the distance between the first and last event in a complex event $A$ is at most 100, i.e.,

$$Q = \{ A \in 2^{\text{tuples}(\mathcal{R})} \mid A \neq \emptyset, \text{first} = \min\{t.ts \mid t \in A\}, \text{last} = \max\{t.ts \mid t \in A\}, \text{first} - \text{last} < 100\},$$

then the evaluation of $\varphi_1$ over such a sliding window is expressed by the formula $\varphi_1'' = (\varphi_1 \text{ IN } W) \text{ FILTER } Q(W)$. In Chapter 5 (Section 5.3.1) we discuss in more detail the difficulties of evaluating queries over user-specified windows, and present a dedicated windowing operator.

Selection strategies operators for CEL will be introduced in Section 3.4.

An operator of CEL that does not occur in Table 3.1, is the anchoring operator $\text{START}$, which specifies that a complex event starts at the beginning of the stream. This feature is not particularly interesting in CER, but we include it as a new operator with the simple objective of capturing the automaton model of Section 3.2. Actually, this operator is intensively used in the context of regular expression programming where an expression of the form "$\wedge R$" marks that $R$ must be evaluated starting from the beginning of the document. Therefore, it is not at all unusual in query languages to include an operator that recognizes events from the beginning of the stream.

3.2. A Computational Model for CEL

In this section, we introduce a formal computational model for evaluating rCEL formulas called complex event automata (CEA for short).
Complex event automata (CEA) extend Finite State Automata (FSA) in several ways. First, CEA are evaluated over streams of infinite length, unlike FSA which are typically evaluated over words of finite length. To handle this, runs of a CEA will actually be computed on finite-length prefixes of the stream. Second, since streams are sequences of tuples and tuples can have infinitely many values, CEA need to deal with an infinite alphabet in contrast to FSA which deal with finite alphabets. To handle this, CEA operate similarly to Symbolic Finite State Automata (Veanes, 2013), which are a form of FSA in which the alphabet is described implicitly by a boolean algebra over the symbols. This allows symbolic FSA to work with a possibly infinite alphabet and, at the same time, use finite state memory for processing the input. CEA work analogously, which is reflected in the transitions being labeled by CEA predicates. Third, CEA need to output complex events, unlike FSA which compute boolean answers. To handle this, CEA operate similarly to Finite State Transducers (Berstel, 2013), which are a form of FSA capable of producing an output whenever an input symbol is read (see below a more detailed comparison between CEA and transducers).

The formal definition of CEA is as follows. Recall from Section 2.2.2 that \( \mathcal{U} \) denotes our universe of FO predicates which contains at least the empty predicate \( \emptyset \); and for every relation symbol \( R \in \mathcal{R} \), the predicate \( \text{tuples}(R) \); and which is closed under union, intersection and difference. As a consequence, \( \mathcal{U} \) also contains the predicate \( \text{tuples}(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} \text{tuples}(R) \), which we will sometimes simply denote by \( \text{TRUE} \) for clarity and emphasis. Recall that \( \mathcal{L} \) is a set of variables.

**Definition 3.** A complex event automaton (CEA) is a tuple \( A = (Q, \Delta, I, F) \) where \( Q \) is a finite set of states, \( \Delta \subseteq Q \times \mathcal{U} \times 2^{\mathcal{L}} \times Q \) is the transition relation, which we require to be finite, and \( I, F \subseteq Q \) are the set of initial and final states, respectively. The size \( |A| \) of \( A \) is its number of states plus its number of edges, \( |A| = |Q| + |\Delta| \). Given a stream \( S = t_0 t_1 \ldots \), a run \( \rho \) of \( A \) over \( S \) is a sequence of transitions: \( \rho : q_0 \xrightarrow{P_0/L_0} q_1 \xrightarrow{P_1/L_1} \ldots \xrightarrow{P_n/L_n} q_{n+1} \) such that \( q_0 \in I \), \( t_i \in P_i \) and \( (q_i, P_i, L_i, q_{i+1}) \in \Delta \) for every \( i \leq n \). We say that \( \rho \) is accepting if \( q_{n+1} \in F \). A position \( 0 \leq i \leq n \) is said to be marked by variable \( A \in \mathcal{L} \) in \( \rho \) if \( A \in L_i \). The position \( i \) is marked if it is marked
by some variable. Similarly, a transition $(q, P, L, p) \in \Delta$ with $L \subseteq L$ is said to be marking if $L \neq \emptyset$.

Just as for CEL, we find it convenient to define two semantics on CEA: a valuation semantics which outputs valuations, and a complex event semantics that outputs complex events. Formally, given a run $\rho : q_0 \xrightarrow{P_0/L_0} q_1 \xrightarrow{P_1/L_1} \cdots \xrightarrow{P_n/L_n} q_{n+1}$ we define the valuation $\mu_\rho$ such that, for every variable $A$, $\mu_\rho(A) = \{0 \leq i \leq n \mid A \in L_i\}$. The complex event associated to $\rho$ is the set of all positions marked by $\rho$.

**Definition 4.** Let $A$ be a CEA and $S$ a stream. Let $\text{Run}_n(A, S)$ denote the set of all accepting runs of $A$ over $S$ that end at position $n \in \mathbb{N}$. The set of valuations of $A$ over $S$ at position $n$ is defined as $[A]_n(S) = \{\mu_\rho \mid \rho \in \text{Run}_n(A, S)\}$. The set of all complex events recognized by $A$ over $S$ at position $n$ is defined as $[J_A]_n(S) = \{\sup(\mu) \mid \mu \in [A]_n(S)\}$.

**Example 1.** Consider as an example the CEA $A$ depicted in Fig. 3.3. In this CEA, the transition $(q_2, \text{tuples}(T) \mid \{T\}, q_2)$ marks one $T$-tuple with a $T$-variable and both transitions labeled by $\text{tuples}(H) \mid \{H_1\}$ and $\text{tuples}(H) \mid \{H_2\}$ mark a $H$-tuple with a $H_1$- and $H_2$-variable, respectively. Note also that the transitions labeled by $\text{TRUE} \mid \emptyset$ allow $A$ to arbitrarily skip tuples of the stream. Then, for every stream $S$, $[A]_n(S)$ represents the set of all complex events at position $n$ that begin and end with an $H$-tuple and that may contain some $T$-tuples between them.

It is important to note that, although CEA operate similarly to transducers (Berstel, 2013) in that they both produce outputs, they are incomparable as computational models. Indeed, a transducer reads strings and produces strings, while a CEA reads the prefix of a stream and produces complex events—which is a set of positions of the
stream. Although transducers could encode a complex event as a string (i.e., the output string has the same length as the input stream prefix, and we represent the output positions with a special symbol), this will be an inefficient representation in practice. In fact, this representation will break the delay guarantees provided by our evaluation algorithm in Section 3.6. Indeed, a complex event consisting of only one position in the stream (e.g., \( \{i\} \)) will be represented by a string of arbitrary size. While our algorithm will produce this complex event in constant time, the transducer will hence need time proportional to the length of the stream prefix to produce the complex event encoding. For this reason, the model of transducers does not apply here, and CEA form a new computational model specially designed for complex event recognition.

3.2.1. I/O-deterministic CEA

Just like standard finite state automata, one may distinguish between deterministic and non-deterministic CEA, in the following sense.

**Definition 5.** A CEA \( \mathcal{A} = (Q, \Delta, I, F) \) is Input/Output deterministic (I/O-deterministic for short) if \( |I| = 1 \) and for any two transitions \((p, P_1, L_1, q_1)\) and \((p, P_2, L_2, q_2)\), either \(P_1\) and \(P_2\) are mutually exclusive (i.e. \(P_1 \cap P_2 = \emptyset\)), or \(L_1 \neq L_2\).

Intuitively, this notion imposes that given a stream \(S\) and a valuation \(\mu\), there is at most one run over \(S\) that generates \(\mu\) (thus the name referencing the input and the output). In contrast, the classic notion of determinism would allow for at most one run over the entire stream. As an example, one can check that the CEA depicted in Fig.3.3 is I/O-deterministic.

The subclass of I/O-deterministic CEA is important because it allows for a simple and efficient evaluation algorithm, as we will see in Section 3.6.

We next show that we can always convert a CEA into an equivalent I/O deterministic one. Formally, we say that two CEA \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are valuation equivalent (denoted \( \mathcal{A}_1 \equiv_v \mathcal{A}_2 \)) if for every stream \(S\) and every index \(n\) we have \( \|\mathcal{A}_1\|_n(S) = \|\mathcal{A}_2\|_n(S) \). We say that \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are complex event equivalent (denoted \( \mathcal{A}_1 \equiv_c \mathcal{A}_2 \)) if for every stream \(S\) and every index \(n\) we have \( \mathcal{J}_n(S) = \mathcal{J}_n(S) \). Clearly, \( \mathcal{A}_1 \equiv_v \mathcal{A}_2 \) implies \( \mathcal{A}_1 \equiv_c \mathcal{A}_2 \) but the converse does not necessarily hold.
PROPOSITION 1. The class of CEA is closed under I/O-determinization: for every CEA $A$ there is a valuation-equivalent I/O-deterministic CEA $A'$ whose size is at most exponential in $|A|$.

The proof requires the following notion.

DEFINITION 6. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\} \subseteq \mathcal{U}$ be a finite set of CEA predicates. Define, for $S \subseteq \{1, \ldots, n\}$, the new CEA predicate

$$P_S = \bigcap_{i \in S} P_i \setminus \bigcup_{i \notin S} P_i.$$ 

Then the set of types of $\mathcal{P}$ is the set of CEA predicates $\text{types}(\mathcal{P}) = \{P_S \mid S \subseteq \{1, \ldots, n\}\}$.

Observe that, by definition, for every tuple $t$ there is at most one predicate $P \in \text{types}(\mathcal{P})$ with $t \in P$. As such, predicates in $\text{types}(\mathcal{P})$ are pairwise mutually exclusive. Also note that, because $\mathcal{U}$ is closed under union, intersection, and difference, every predicate in $\text{types}(\mathcal{P})$ is also in $\mathcal{U}$. We are now ready prove Proposition 1.

PROOF OF PROPOSITION 1. The construction is similar to the one used for classical NFA determinization: $A'$ will maintain, in its states, the set of states that $A$ is at in any run. The difficulty in our case comes from the fact that in a given state and reading an event $t$, there may be many transitions of $A$—all with different predicates—that can read $t$. How to encode all of these transitions by single transition—and hence one predicate—in $A'$? The solution is given by $\text{types}(\mathcal{P})$, which partitions the set of all tuples in a way that if a tuple $t$ satisfies a predicate $P_t \in \text{types}(\mathcal{P})$, then $P_t$ is a subset of the predicates of all transitions that a run of $A$ could take when reading $t$.

Formally, consider an arbitrary CEA $A = (Q, \Delta, I, F)$. Let $\mathcal{P}$ and $\mathcal{L}$ be the set of all predicates and variable sets, respectively, occurring in transitions in $\Delta$. For convenience, we extend the transition relation $\Delta$ as a function such that:

$$\Delta(R, P, L) = \{q \in Q \mid \exists p \in R, P' \in \mathcal{P} : P \subseteq P' \land (p, (P', L), q) \in \Delta\},$$
for every \( R \subseteq Q, P \in \text{types}(\mathcal{P}) \), and \( L \in \mathcal{L} \). We then define the CEA \( \mathcal{A}' = (Q', \Delta', I', F') \) as follows. First, the set of states is \( Q' = 2^Q \), that is, each state in \( Q' \) represents a set of states of \( Q \). Second, the transition relation consists of all transitions \( (R, P, L, U) \) such that \( R \in Q', P \in \text{types}(\mathcal{P}), L \in \mathcal{L} \), and \( U = \Delta(R, P, L) \). Finally, the sets of initial and final states are \( I' = \{ I \} \) and \( F' = \{ T \in Q' \mid T \cap F \neq \emptyset \} \). Notice that \( \mathcal{A}' \) is I/O-deterministic because predicates in \( \text{types}(\mathcal{P}) \) are pairwise disjoint by construction.

The fact that \( \|\mathcal{A}\|_n(S) = \|\mathcal{A}'\|_n(S) \) for every stream \( S \) and index \( n \) now follows similarly as for classical NFA determinization: namely, an accepting run of length \( n \) in \( \mathcal{A} \) can be translated into an accepting run in \( \mathcal{A}' \) where the set-states contain the states from the original run. On the other hand, an accepting run in \( \mathcal{A}' \) can only exist if a sequence of states using the same transitions exists in the original automaton \( \mathcal{A} \). Finally, note that \( \mathcal{A}' \) has \( 2^{|Q|} \) states and mentions at most \( 2^{|\mathcal{P}|} \) predicates. Therefore, writing \( \mathcal{P}' \) for \( \text{types}(\mathcal{P}) \) we have:

\[
|\mathcal{A}'| = |Q'| + |\Delta'| \leq |Q'| + |Q'| \times |\mathcal{P}'| \times |Q'| \times |\mathcal{L}|
= O(|Q| \times |\mathcal{P}'| \times |Q| \times |\mathcal{L}|)
= O(2^{|Q|} \times 2^{|\mathcal{P}|} \times 2^{|Q|} \times 2^{|\mathcal{L}|})
= O(2^{2|Q|+|\Delta|})
= O(2^{2|\mathcal{A}|}).
\]

The size of \( \mathcal{A}' \) is hence exponential in \( |\mathcal{A}| \). \( \square \)

### 3.3. Comparing Unary rCEL and CEA

In this section, we show that unary rCEL and CEA are expressively equivalent. We first show how to translate unary rCEL into CEA, and then conversely how to translate CEA into unary rCEL. As a by-product of these translations, we obtain that several operators are not primitive in unary rCEL, and can be expressed using only variable binding, filtering, contiguous sequencing and iteration, disjunction, projection, and anchoring.
DEFINITION 7. A CEL formula $\varphi$ and a CEA $A$ are valuation equivalent if, for every $S$ and $n$, it is the case that $[\varphi]_n(S) = [A]_n(S)$. They are complex event equivalent if $[\varphi]_n(S) = [A]_n(S)$ for every $S$ and $n$.

Clearly, valuation equivalence implies complex event equivalence, but not conversely.

THEOREM 1. If $\varphi$ is a unary rCEL formula, then there exists a CEA $A_\varphi$ that is evaluation equivalent with $\varphi$.

PROOF. For this proof, it will be useful to slightly extend the semantics of CEA so that a CEA can be run on an arbitrary interval of a stream, instead of always starting at position 0. Specifically, let $S = t_0t_1 \ldots$ be a stream and $A = (Q, \Delta, I, F)$ be a CEA. For a valuation $\mu$ and $i \in \mathbb{N}$ define the valuation $\mu^{+i}$ such that $\mu^{+i}(A) = \{k + i \mid k \in \mu(A)\}$ for every $A \in \mathcal{L}$. Recall that $S_i = t_it_{i+1} \ldots$. Then we define the set of valuations of $A$ over $S$ from positions $i$ to $j$ as $[A]^{+i}(S, i, j) = \{\mu^{+i} \mid \mu \in [A]^{j-\ell}(S_i)\}$. In other words, $[A]^{+i}(S, i, j)$ computes all the valuations obtained from the evaluation of $A$ from positions $i$ to $j$.

To obtain the theorem we prove the following stronger property: For every unary rCEL formula $\varphi$ there exists a CEA $A_\varphi$ such that

$$[\varphi]^{+i}(S, i, j) = [A_\varphi]^{+i}(S, i, j), \text{ for every stream } S \text{ and positions } i \leq j. \quad (3.1)$$

The construction of $A_\varphi$ is by induction on $\varphi$.

- If $\varphi = R$, then $A_\varphi$ is defined as in Fig. 3.4: $A_\varphi = (\{q_1, q_2\}, \Delta_\varphi, \{q_1\}, \{q_2\})$ with $\Delta_\varphi = \{(q_1, \text{tuples}(R), \{R\}, q_2), (q_1, \text{TRUE}, \emptyset, q_1)\}$.

- If $\varphi = \psi \text{ IN } A$, then $A_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where $\Delta_\varphi$ is the result of adding variable $A$ to all marking transitions of $\Delta_\psi$. Formally, $\Delta_\varphi = \{(p, P, L, q) \in$
• If $\varphi = \psi \oplus$ FILTER $P^{SO}(A)$ for some CEA predicate $P \in \mathcal{U}$ and some variable $A \in \mathbf{L}$, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where $\Delta_\varphi$ is defined as $\{(p, P', L, q) \in \Delta_\psi \mid A \notin L\} \cup \{(p, P \land P', L, q) \mid (p, P', L, q) \in \Delta_\psi \land A \in L\}$. The intuition behind this is that since $P^{SO}$ is the second order extension of $P$, all tuples that are labeled by $A$ must satisfy $P$.

• If $\varphi = \psi_1 \oplus \psi_2$, then $\mathcal{A}_\varphi = (Q_1 \cup Q_2, \Delta_\varphi, I_\psi, F_\psi)$ where $\Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \cup \{(p, P, L, q) \mid q \in I_{\psi_2} \land \exists q' \in F_{\psi_1}, (p, P, L, q') \in \Delta_{\psi_1}\}$. Here, we assume w.l.o.g. that $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ have disjoint sets of states.

• If $\varphi = \psi^+$, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where $\Delta_\varphi = \Delta_\psi \cup \{(p, P, L, q) \mid q \in I_\psi \land \exists q' \in F_\psi, (p, P, L, q') \in \Delta_\psi\}$.

• If $\varphi = \psi_1 : \psi_2$, then we do the following. In order to obtain the contiguous sequencing semantics, we will construct $\mathcal{A}_\varphi$ by connecting $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ through a new fresh state $q$. We ensure that all transitions arriving at or departing from $q$ mark at least one variable. As such, any accepting run of $\mathcal{A}_\psi$ that generates a valuation $\mu$ will be able to be decomposed into accepting runs of $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ generating valuations $\mu_1$ and $\mu_2$ with $\max(\sup(\mu_1)) + 1 = \min(\sup(\mu_2))$. Formally, we define $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ as follows. First, the set of states is $Q_\varphi = Q_\psi \cup Q_\varphi \cup \{q\}$, where $q$ is a new fresh state. Then, the transition relation is $\Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \cup \{(q_1, P, L, q) \mid L \neq \emptyset \land \exists q' \in F_{\psi_1}, ((q_1, P, L, q') \in \Delta_{\psi_1}) \cup \{(q, P, L, q_2) \mid L \neq \emptyset \land \exists q' \in I_{\psi_2}, ((q', P, L, q_2) \in \Delta_{\psi_2})\}$. Finally, the sets of initial and final states are $I_\varphi = I_{\psi_1}$ and $F_\varphi = F_{\psi_2}$.

• If $\varphi = \psi \oplus$, then we can use an idea similar to the previous case. We add a new fresh state $q$, which will make the connection between one iteration and the next one. In order to obtain the $\oplus$ semantics, we will restrict $q$ so that only transitions labeled with a non-empty set of variables arrive at and depart from $q$. We do this as follows: we define $\mathcal{A}_\varphi = (Q_\psi \cup \{q\}, \Delta_\varphi, I_\psi, F_\psi)$. The transition relation is $\Delta_\varphi = \Delta_\psi \cup \{(q_1, P, L, q) \mid L \neq \emptyset \land \exists q' \in F_{\psi_1}, ((q_1, P, L, q') \in \Delta_{\psi_1}) \cup \{(q, P, L, q_2) \mid L \neq \emptyset \land \exists q' \in I_{\psi_2}, ((q', P, L, q_2) \in \Delta_{\psi_2})\}$. Finally, the sets of initial and final states are $I_\varphi = I_{\psi_1}$ and $F_\varphi = F_{\psi_2}$.
\(\emptyset \land \exists q' \in F_\psi \cdot ((q_1, P, L, q') \in \Delta_\psi) \cup \{(q, P, L, q_2) \mid L \neq \emptyset \land \exists q' \in I_\psi \cdot ((q', P, L, q_2) \in \Delta_\psi)\}.

- If \(\varphi = \pi_L(\psi)\) for some \(L \subseteq \mathbf{L}\), then \(\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\varphi, F_\psi)\) where \(\Delta_\varphi\) is the result of intersecting the labels of each transition in \(\Delta_\psi\) with \(L\). Formally, that is \(\Delta_\varphi = \{(p, P, L \cap L', q) \mid (p, P, L', q) \in \Delta_\psi\}\).

- If \(\varphi = \text{START}(\psi)\), then we need to force the first transition to mark at least one variable. We do so by adding a new fresh state \(q\) which will work as our initial state, and enforce that all departing transitions are labeled with a non-empty set of variables. Formally, \(\mathcal{A}_\varphi = (Q_\psi \cup \{q\}, \Delta_\varphi, \{q\}, F_\psi)\), where \(\Delta_\varphi = \Delta_\psi \cup \{(q, P, L, p) \mid L \neq \emptyset \land \exists q' \in I_\psi \cdot ((q', P, L, p) \in \Delta_\psi)\}\).

- If \(\varphi = \psi_1 \text{ OR } \psi_2\), then \(\mathcal{A}_\varphi\) is essentially the automata union between \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\) as one would expect: \(\mathcal{A}_\varphi = (Q_{\psi_1} \cup Q_{\psi_2}, \Delta_{\psi_1} \cup \Delta_{\psi_2}, I_{\psi_1} \cup I_{\psi_2}, F_{\psi_1} \cup F_{\psi_2})\).

Here, we assume w.l.o.g. that \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\) have disjoint sets of states.

- If \(\varphi = \psi_1 \text{ AND } \psi_2\), then \(\mathcal{A}_\varphi\) is the product automaton between \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\) as one would expect: \(\mathcal{A}_\varphi = (Q_{\psi_1} \times Q_{\psi_2}, \Delta_{\varphi}, I_{\psi_1} \times I_{\psi_2}, F_{\psi_1} \times F_{\psi_2})\) where \(\Delta_{\varphi} = \{(q_1, q_2), P_1 \cap P_2, L, (p_1, p_2)) \mid (q_1, P_1, L, p_1) \in \Delta_{\psi_1} \land (q_2, P_2, L, p_2) \in \Delta_{\psi_2}\}\).

- If \(\varphi = \psi_1 \text{ \textsc{ALL} } \psi_2\), then assume that the initial states in \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\) have no incoming transitions, and the final states in \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\) have no outgoing transitions. This is without loss of generality since when an initial state \(q\) has incoming transitions then we may always make a (non-initial) copy \(q'\) of \(q\) (copying all outgoing transitions) and re-direct all incoming transitions of \(q\) to \(q'\) instead. This removes the incoming transitions from \(q\) while preserving value-equivalence. A similar transformation works for the final states.

\(\mathcal{A}_\varphi\) is now again a form of product automaton between \(\mathcal{A}_{\psi_1}\) and \(\mathcal{A}_{\psi_2}\). To obtain the semantics of \(\text{\textsc{ALL}}\) this product automaton will allow a sub-automaton (i.e., \(\mathcal{A}_{\psi_1}\) or \(\mathcal{A}_{\psi_2}\)) to start later or stop earlier than the other sub-automaton in the run. Formally, \(\mathcal{A}_\varphi = (Q_{\psi_1} \times Q_{\psi_2}, \Delta_{\varphi}, I_{\psi_1} \times I_{\psi_2}, F_{\psi_1} \times F_{\psi_2})\) where \(\Delta_{\varphi}\) is defined as follows.
\[ \Delta_\varphi = \{ ((q_1, q_2), P_1 \cap P_2, L_1 \cup L_2, (p_1, p_2)) \mid (q_1, P_1, L_1, p_1) \in \Delta_{\psi_1} \]

\[ \land (q_2, P_2, L_2, p_2) \in \Delta_{\psi_2} \} \]

\[ \cup \{(q_1, q_2), P_2, L_2, (q_1, p_2)) \mid q_1 \in I_{\psi_1}, (q_2, P_2, L_2, p_2) \in \Delta_{\psi_2} \} \]

\[ \cup \{(q_1, q_2), P_1, L_1, (p_1, q_2)) \mid q_2 \in I_{\psi_2}, (q_1, P_1, L_1, p_1) \in \Delta_{\psi_1} \} \]

\[ \cup \{(q_1, q_2), P_2, L_2, (q_1, p_2)) \mid q_1 \in F_{\psi_1}, (q_2, P_2, L_2, p_2) \in \Delta_{\psi_2} \} \]

\[ \cup \{(q_1, q_2), P_1, L_1, (p_1, q_2)) \mid q_2 \in F_{\psi_2}, (q_1, P_1, L_1, p_1) \in \Delta_{\psi_1} \}. \]

Here, the first set of transitions of the union lets \( \mathcal{A}_\varphi \) simulate \( \mathcal{A}_{\psi_1} \) and \( \mathcal{A}_{\psi_2} \) in lock-step, while producing a valuation that is the union of the valuations produced by \( \mathcal{A}_{\psi_1} \) and \( \mathcal{A}_{\psi_2} \). The other sets of transitions allow a sub-automaton to start later, or finish earlier, than the other. The assumption that initial states are without incoming transitions and final states are without outgoing transitions is necessary because otherwise \( \mathcal{A}_\varphi \) could produce runs where, for example, it first runs \( \mathcal{A}_{\varphi_1} \) alone (\( \mathcal{A}_{\varphi_2} \) loops in an initial state), then \( \mathcal{A}_{\varphi_1} \) and \( \mathcal{A}_{\varphi_2} \) in lock-step, then \( \mathcal{A}_{\varphi_1} \) alone again (because \( \mathcal{A}_{\varphi_2} \) re-transitioned to one of its initial states, and now loops there), and then back to simulating \( \mathcal{A}_{\varphi_1} \) and \( \mathcal{A}_{\varphi_2} \) in lock-step. Such behavior is inconsistent with the \textit{ALL} semantics because the two subruns of \( \mathcal{A}_{\varphi_2} \) need not be combinable to a single subrun of \( \mathcal{A}_{\varphi_2} \). Similar problems may occur if final states have outgoing transitions.

- If \( \varphi = \psi_1 \text{ UNLESS } \psi_2 \) then we first construct automaton \( \mathcal{E} = (Q_\mathcal{E}, \Delta_\mathcal{E}, I_\mathcal{E}, F_\mathcal{E}) \) for the formula \( \psi'_{\mathcal{E}} = \pi_\emptyset(\psi_2) \), and I/O determinize it. Note that, by the translation of \( \pi_\emptyset \) described above, every transition in \( \Delta_\mathcal{E} \) will only mark the empty set of variables. Therefore, every accepting run of \( \mathcal{E} \) yields the empty valuation (i.e., the valuation that maps every variable to the empty complex event). Since there is only one possible such valuation, and the automaton is I/O deterministic, it is actually deterministic: on every \((S, i, j)\) there is at most one accepting run. Then \( \mathcal{A}_{\varphi} \) is obtained by simulating \( \mathcal{A}_{\varphi_1} \) and \( \mathcal{E} \) in conjunction, but only accepting when \( \mathcal{A}_{\psi_1} \) accepts and \( \mathcal{E} \) rejects.
Formally, $\mathcal{A}_\varphi = (Q_{\psi_1} \times Q_{\varepsilon}, \Delta_\varphi, I_{\psi_1} \times I_{\varepsilon}, F_{\psi_1} \times (Q_{\varepsilon} \setminus F_{\varepsilon}))$ where $\Delta_\varphi = \{((q_1, q_2), P_1 \cap P_2, (p_1, p_2)) \mid (q_1, P_1, L, p_1) \in \Delta_{\psi_1}, (q_2, P_2, \emptyset, p_2) \in \Delta_{\varepsilon}\}$.

Property (3.1) is now obtained by induction on $\varphi$.  

**Theorem 2.** For every CEA $A$, there exists a valuation-equivalent unary rCEL formula $\varphi_A$ that does not use any operator in $\{; , +, \text{AND}, \text{ALL}, \text{UNLESS}\}$.  

**Proof.** Let $A = (Q, \Delta, I, F)$ be a CEA with $Q = \{q_1, \ldots, q_n\}$. Assume that there is only one initial state and one final state, i.e., $I = \{q_1\}$ and $F = \{q_n\}$. This is without loss of generality: we can always ensure a single initial state by adding a new initial state and copying each out-transition of the original initial states as an out-transition of the new initial state; likewise, we can ensure a single final state by adding a new final state and copying all in-transitions of the original final states as an in-transition of the new final state.

The main idea is based on the construction used to convert standard FSA into regular expressions. We define, for every pair of states $q_i, q_j$, a formula $\varphi_{ij}$ that represents the complex events defined by the runs from $q_i$ to $q_j$. To aid in the definition of $\varphi_{ij}$ we will first define, for every $1 \leq k \leq |Q|$ the formula $\varphi_{ij}^k$ that represents the valuations defined by the runs from $q_i$ to $q_j$ that only visit states in $\{q_1, \ldots, q_k\}$. It is then clear that $\varphi_{ij}^{|Q|} = \varphi_{ij}$.

We define $\varphi_{ij}^k$ recursively as follows. In the base case $k = 0$, for each $i, j$, if there is no transition from $q_i$ to $q_j$ in $A$, then $\varphi_{ij}^0 = \text{FALSE}$ where $\text{FALSE}$ is a formula that is never satisfied. One way to define it is $\text{FALSE} = (R \text{ FILTER } \emptyset)$. If there is at least one transition from $q_i$ to $q_j$ in $A$, we do the following. For convenience, if $L = \{A_1, \ldots, A_l\}$ is a non-empty set of variables and $\psi$ is a rCEL formula, let us simply write $\psi \text{ IN } L$ for the more verbose $(\ldots (\psi \text{ IN } A_1) \text{ IN } A_2) \text{ IN } \ldots) \text{ IN } A_l$. Assume that $\mathcal{R} = \{R_1, \ldots, R_r\}$. Define, for any transition $t = (q_i, P, L, q_j)$ of $A$ the formula $\psi_t$ by

$$\psi_t = \pi_L[(R_1 \text{ OR } \cdots \text{ OR } R_r) \text{ IN } (L \cup \{A\}) \text{ FILTER } P^{SO}(A)],$$

where $\pi_L$ is the projection onto the variables in $L$. Then $\varphi_{ij}^k$ is defined as

$$\varphi_{ij}^k = \begin{cases} \text{FALSE} & \text{if } k = 0 \\ \varphi_{ij}^{k-1} \text{ FILTER } P^{SO}(A) & \text{if } k > 0 \end{cases}$$

where $P^{SO}(A)$ is the projection of the path in $A$ to the variables in $L$. Finally, $\varphi_{ij}^{|Q|}$ is defined as $\varphi_{ij}^{|Q|} = \varphi_{ij}$.  

Property (3.1) is now obtained by induction on $\varphi$.  

□
where \( A \) is a “fresh” variable, distinct from the relation names in \( R \) and any variable mentioned in \( \mathcal{A} \). Intuitively, \( \psi_t \) simulates transition \( t \) by detecting any event (of any type), marking the event with all variables in \( L \) as well as \( A \), checking that \( P \) holds for the event, and projecting on \( L \) to ensure that the event is marked with the same variables as those of \( t \). Then we define

\[
\varphi_{ij}^0 = \psi_{t_1} \lor \ldots \lor \psi_{t_m},
\]

where \( t_1, \ldots, t_m \) are all transitions from \( q_i \) to \( q_j \) in \( \mathcal{A} \). Next, for \( k > 0 \) the recursion is defined as:

\[
\varphi_{ij}^k = \varphi_{ij}^{k-1} \lor (\varphi_{ik}^{k-1} : \varphi_{kj}^{k-1}) \lor (\varphi_{ik}^{k-1} : \varphi_{kk}^{k-1} \oplus : \varphi_{kj}^{k-1})
\]

The final formula \( \varphi_A \) is the result of considering \( \varphi_{1n} \), forcing it to begin immediately at position 0: \( \varphi_A := \text{START}(\varphi_{1n}) \). Correctness of this construction can now be proved by induction over the number of states.

By combining theorems 1 and 2 we obtain that \{ ; , +, AND , ALL , UNLESS \} are expressively redundant in unary rCEL.

**Corollary 1.** For every unary rCEL formula there exists a valuation-equivalent rCEL formula constructed only from operators in \{ IN , FILTER , :, \oplus , \pi , START , OR \}.

### 3.4. Selection Strategies

For the purpose of illustration, consider the following running example.

**Example 2.** Assume there is a stream produced by wireless sensors positioned in a farm, whose main objective is to detect fires. As a first scenario, assume that there are three sensors, and each of them can measure both temperature (in Celsius degrees) and relative humidity (as the percentage of vapor in the air). Each sensor is assigned

<table>
<thead>
<tr>
<th>type</th>
<th>H</th>
<th>T</th>
<th>H</th>
<th>T</th>
<th>T</th>
<th>H</th>
<th>H</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>value</td>
<td>25</td>
<td>45</td>
<td>20</td>
<td>25</td>
<td>40</td>
<td>42</td>
<td>25</td>
<td>70</td>
</tr>
</tbody>
</table>

**Figure 3.5.** A stream \( S \) of events measuring temperature and humidity. \( \text{“value”} \) contains degrees and humidity for \( T \)- and \( H \)- events, respectively.
an id in \{0, 1, 2\}. The events produced by the sensors consist of the id of the sensor and a measurement of temperature or humidity. In favor of brevity, we write $T(id, tmp)$ for an event reporting temperature $tmp$ from sensor with id $id$, and similarly $H(id, hum)$ for events reporting humidity. Figure 3.5 depicts such a stream: each column is an event and the value row is the temperature or humidity if the event is of type $T$ or $H$, respectively.

For the sake of illustration, assume that the position of sensor 0 is particularly prone to fires, and it has been detected that a temperature measurement above 40 degrees Celsius followed by a humidity measurement of less than 25% represents a fire with high probability. The following formula then represents such pattern:

$$\rho_1 := (T ; H) \text{FILTER} (T.tmp > 40 \land H.hum <= 25 \land T.id = 0 \land H.id = 0).$$

We may sometimes also be interested in sequences of $T$ and $H$ where the $H$ event immediately follows the $T$ event. For such situations, CEL is also endowed with a contiguous version of sequencing (:). The following variant $\varphi_1'$ of $\varphi_1$ hence only returns \{1, 2\} on the stream in Fig. 1.1.

$$\rho_1' := (T : H) \text{FILTER} (T.tmp > 40 \land H.hum <= 25 \land T.id = 0 \land H.id = 0).$$

Next, assume that we want to see how temperature changes in the location of sensor 1 when there is an increase of humidity. A problem here is that we do not know a priori the amount of temperature measurements; we need to capture an unbounded amount of events. The iteration operator $+$ (a.k.a. Kleene closure) (Cugola & Margara, 2012b; Arasu et al., 2003; Gyllstrom, Agrawal, Diao, & Immelman, 2008) is introduced in most CER frameworks for solving this problem. This operator introduces many difficulties in the semantics of CER languages. For example, since events are not required to occur contiguously, the nesting of $+$ is particularly tricky and most frameworks simply disallow this (see (Wu et al., 2006; Arasu, Babu, & Widom, 2006; Demers et al., 2006)). Coming back to our example, the formula for measuring temperatures
whenever an increase of humidity is detected by sensor 1 is:

\[ \rho_3 := \left[ H \text{ in } H_1 \land T+ \land H \text{ in } H_2 \right] \]

\[ \text{FILTER} \left( H_1.hum < 30 \land H_2.hum > 60 \land H.id = 1 \land T.id = 1 \right). \]

Matching complex events is a computationally intensive task, especially when non-contiguous sequencing and iteration are used. Not all of the produced complex events may actually be useful for the user. For instance, reconsider formula \( \rho_1 \) from Example 2. It selects all pairs of \( T \) and \( H \) events produced by sensor 0, where \( T \)'s temperature is above 40, \( H \)'s humidity is below 25, and \( H \) follows \( T \) — no matter how large the difference in time is between the \( T \) event and the \( H \) event. In particular, \( \rho_1 \) outputs complex events \( \{1, 2\} \), \( \{1, 8\} \), and \( \{5, 8\} \) when evaluated on stream \( S \) of Figure 3.5. One may argue, however, that, while the \( H \) event need not strictly follow the \( T \) event (due to noisy streams), there is no point in generating a new complex event for a new \( H \) event when a \( T \) event was already previously output. I.e., that while \( \{1, 2\} \) is a reasonable output, \( \{1, 8\} \) is not. If we remove outputs like \( \{1, 8\} \) from the result we actually also gain in processing efficiency: an event that has already been successfully output need not be considered against future events for possible additional matches.

In the CER literature, it is common to apply so-called selection strategies (or selectors) (Cugola & Margara, 2012b) to a CER pattern to restrict the set of results (Carlson & Lisper, 2010; Wu et al., 2006; Zhang et al., 2014) and speed up query processing. Unfortunately, however, most proposals in the literature introduce selection strategies as heuristics that apply to particular computational models without describing how the semantics are affected.

In this section, we present a proposal on how to formalize selection strategies at the semantic level, as unary operators over CEL formulas. We define four selection strategies: strict (STRICT), next (NXT), last (LAST) and max (MAX). STRICT and NXT are motivated by the strict-contiguity and skip-till-next-match selector strategies proposed by SASE (Gyllstrom et al., 2008), while, as we will argue, LAST and MAX are useful selection strategies from a semantic viewpoint. We define each selection strategy below, giving the motivation and formal semantics. Interestingly, we show that for unary CEL
these selection strategies does not add expressive power. In particular, any of these sele-
ction strategies can be compiled into CEA, whose expressive power is equivalent to
unary CEL.

3.4.1. The semantics of common selection strategies in CER

3.4.1.1. STRICT

As the name suggest, STRICT or strict-contiguity keeps only the complex events
that are contiguous in the stream. To motivate this, recall that formula $\rho_1$ in Exam-
ple 2 detects complex events composed by a temperature above 40 degrees followed
by a humidity of less than 25%. As already argued, in general one could expect other
events between $x$ and $y$. However, it could be the case that this pattern is of interest
only if the events occur contiguously in the stream, or perhaps the stream has been pre-
processed by other means and irrelevant events have been thrown out already. For this
purpose, STRICT reduces the set of outputs by selecting only strictly consecutive com-
plex events. Formally, for any CEL formula $\varphi$ we have that $\mu \in \llbracket \text{STRICT}(\varphi) \rrbracket (S, i, j)$
holds if $\mu \in \llbracket \varphi \rrbracket (S, i, j)$ and for every $k_1, k_2 \in \sup(\mu)$ and $k \in \mathbb{N}$, if $k_1 < k < k_2$ then
$k \in \sup(\mu)$ (i.e., $\sup(\mu)$ is an interval). In our running example, $\text{STRICT}(\rho_1)$ would
only produce $\{1, 2\}$, although $\{1, 8\}$ and $\{5, 8\}$ are also outputs for $\rho_1$ over $S$. In other
words, $\text{STRICT}(\rho_1)$ is equivalent to $\rho'_1$. We will have more to say about the relationship
between STRICT and the contiguous sequencing ($:\_:$) and iteration ($\oplus$) operators in
Section 3.5.2.

3.4.1.2. NXT

The second selector, NXT, is similar to the skip-till-next-match operator proposed
in (Gyllstrom et al., 2008). The motivation behind this operator comes from a heuristic
that consumes a stream by skipping those events that cannot participate in the output,
while matching patterns in a greedy manner that selects only the first event satisfy-
ing the next element of the query. In (Gyllstrom et al., 2008) the following informal
definition for this strategy is given:
“a further relaxation is to remove the contiguity requirements: all irrelevant events will be skipped until the next relevant event is read” (*).

In practice, skip-till-next-match is defined by an evaluation algorithm that greedily adds an event to the output whenever a sequential operator is used, or adds as many events as possible whenever an iteration operator is used. The fact that the semantics is only defined by an algorithm requires a user to understand the algorithm to write meaningful queries. In other words, this operator speeds up the evaluation by sacrificing the clarity of the semantics.

To overcome the above problem, we formalize the intuition behind (*) based on a special order over complex events, which we denote by \( \leq_{\text{next}} \). Let \( C_1 \) and \( C_2 \) be complex events. The symmetric difference between \( C_1 \) and \( C_2 \), denoted \( C_1 \triangle C_2 \), is the set of all elements either in \( C_1 \) or \( C_2 \) but not in both. We define \( C_1 \leq_{\text{next}} C_2 \) if either \( C_1 = C_2 \) or \( \min(C_1 \triangle C_2) \in C_2 \). For example, \( \{5, 8\} \leq_{\text{next}} \{1, 8\} \) since the minimum element in \( \{5, 8\} \triangle \{1, 8\} = \{1, 5\} \) is 1, which is in \( \{1, 8\} \). Note that, intuitively, the definition of \( \leq_{\text{next}} \) is similar to skip-till-next-match since \( C_1 \leq_{\text{next}} C_2 \) if \( C_2 \) contains an event that precedes (in stream order) an event in \( C_1 \), but which was skipped in \( C_1 \). In other words: \( C_2 \) selected the first relevant event.

An important property is that the \( \leq_{\text{next}} \)-relation forms a total order among complex events, implying the existence of a minimum and a maximum over any finite set of complex events.

**Lemma 1.** \( \leq_{\text{next}} \) is a total order between complex events.

**Proof.** For \( \leq_{\text{next}} \) to be a total order between complex events, it has to be reflexive (trivial), anti-symmetric, transitive, and total. The proof for each property is given next.

**Anti-symmetric.**

Consider any two complex events \( C_1 \) and \( C_2 \) such that \( C_1 \leq_{\text{next}} C_2 \) and \( C_2 \leq_{\text{next}} C_1 \). Assume, for the purpose of contradiction, that \( C_1 \neq C_2 \). Then because \( C_2 \leq_{\text{next}} C_1 \) and \( C_1 \neq C_2 \) we have \( \min(C_1 \triangle C_2) \in C_1 \). At the same time, because \( C_1 \leq_{\text{next}} C_2 \)
and $C_1 \neq C_2$ we also have $\min(C_1 \triangle C_2) \in C_2$. However, by definition of $C_1 \triangle C_2$, $\min(C_1 \triangle C_2)$ can not be in both $C_1$ and $C_2$, reaching a contradiction. Therefore, $C_1 = C_2$.

**Transitive.**

Consider any three complex events $C_1$, $C_2$ and $C_3$ such that $C_1 \leq_{\text{next}} C_2$ and $C_2 \leq_{\text{next}} C_3$. Obviously, if $C_1 = C_2$ or $C_2 = C_3$ then $C_1 \leq_{\text{next}} C_3$ trivially holds. Hence, assume that $C_1 \neq C_2$ and $C_2 \neq C_3$. Let $l_1 = \min(C_1 \triangle C_2)$ and $l_2 = \min(C_2 \triangle C_3)$. Because $C_1 \leq_{\text{next}} C_2$ and $C_1 \neq C_2$ we have (1) $l_1 = \min(C_1 \triangle C_2) \in C_2$. Because $C_2 \leq_{\text{next}} C_3$ and $C_2 \neq C_3$ we have (2) $l_2 = \min(C_2 \triangle C_3) \in C_3$. Note that, by definition of $C_2 \triangle C_3$ and (2), $l_2 \notin C_2$. As such, $l_1 \neq l_2$.

Next, define for every $i \in \{1, 2, 3\}$ and $j \in \{1, 2\}$ the set $C_{i}^{<l_j}$ as the set of elements of $C_i$ which are lower than $l_j$, i.e., $C_{i}^{<l_j} = \{x \mid x \in C_i \land x < l_j\}$. It is clear that $C_{1}^{<l_1} = C_{2}^{<l_2}$ and $C_{2}^{<l_2} = C_{3}^{<l_3}$, because of (1) and (2), respectively.

Consider first the case where $l_1 < l_2$. This means that (3) $C_{1}^{<l_1} = C_{3}^{<l_3}$. Moreover, if $l_1$ were not in $C_3$, it would contradict (2), so (4) $l_1 \in C_3$ must hold. With (3) and (4), it follows that $l_1$ is the lowest element that is either in $C_1$ or $C_3$ but not in both, and it is in $C_3$. This proves that $\min(C_1 \triangle C_3) \in C_3$, and thus $C_1 \leq_{\text{next}} C_3$.

Now consider the case where $l_2 < l_1$. Then, (5) $C_{1}^{<l_2} = C_{3}^{<l_2}$ must hold. Because $l_2$ is not in $C_2$, it cannot be in $C_1$, otherwise it would contradict (1), so (6) $l_2 \notin C_1$ must hold. Also, because of (2) we know that (7) $l_2 \in C_3$ must hold. With (5), (6) and (7), it follows that $l_2$ is the lowest element that is either in $C_1$ or $C_3$ but not in both, and it is in $C_3$. This proves that $\min(C_1 \triangle C_3) \in C_3$, and thus $C_1 \leq_{\text{next}} C_3$.

**Total.**

Consider any two complex events $C_1$ and $C_2$. If $C_1 = C_2$, then $C_1 \leq_{\text{next}} C_2$ holds. Consider now the case where $C_1 \neq C_2$. Define the set $C = C_1 \triangle C_2$. Because $C_1 \neq C_2$, there must be at least one element in $C$. In particular, this implies that there is a minimum element $l$ in $C$. If $l$ is in $C_2$, then $C_1 \leq_{\text{next}} C_2$ holds, and if $l$ is in $C_1$, then $C_2 \leq_{\text{next}} C_1$ holds. □
We can define now the semantics of \( \text{NXT} \): for a CEL formula \( \varphi \) we have that 
\[
\mu \in \llbracket \text{NXT}(\varphi) \rrbracket(S, i, j) \text{ if } \mu \in \llbracket \varphi \rrbracket(S, i, j) \text{ and } \sup(\mu') \leq_{\text{next}} \sup(\mu) \text{ for every } \mu' \in \llbracket \varphi \rrbracket(S, i, j).
\] In other words, \( \sup(\mu) \) must be the \( \leq_{\text{next}} \)-maximum complex event among all complex events within the same interval. In our running example, we have that \( \{1, 8\} \) is in \( \llbracket \text{NXT}(\rho_1) \rrbracket(S) \) but \( \{5, 8\} \) is not. Furthermore, \( \{3, 4, 6, 7\} \in \llbracket \text{NXT}(\rho_3) \rrbracket(S) \) while \( \{3, 4, 7\} \) and \( \{3, 6, 7\} \) are not.

### 3.4.1.3. LAST

The \( \text{NXT} \) selector is motivated by the computational benefit of skipping irrelevant events in a greedy fashion. However, from a semantic point of view it might not be what a user wants. For example, if we consider again \( \rho_1 \) and the stream in Figure 3.5, we know that every complex event in \( \text{NXT}(\rho_1) \) will have event 1. In this sense, the \( \text{NXT} \) strategy selects the oldest complex event for the formula. We argue here that a user might actually prefer the opposite, i.e. the most recent explanation for the matching of a formula. This is the idea captured by \( \text{LAST} \). Formally, the \( \text{LAST} \) selector is defined exactly as \( \text{NXT} \), but changing the order \( \leq_{\text{next}} \) by \( \leq_{\text{last}} \): if \( C_1 \) and \( C_2 \) are two complex events, then \( C_1 \leq_{\text{last}} C_2 \) if either \( C_1 = C_2 \) or \( \max(C_1 \triangle C_2) \in C_2 \). For example, \( \{1, 8\} \leq_{\text{last}} \{5, 8\} \). In our running example, \( \text{LAST}(\rho_1) \) would select the most recent temperature and humidity that explain the matching of \( \rho_1 \) (i.e. \( \{5, 8\} \)), which might be a better explanation for a possible fire. Surprisingly, as we will see \( \text{LAST} \) enjoys the same computational properties as \( \text{NXT} \), even though it does not come from a greedy heuristic like \( \text{NXT} \) does.

### 3.4.1.4. MAX

A more ambitious selection strategy is to keep the maximal complex events in terms of set inclusion, which could be naturally more useful because these complex events are the most informative. Formally, given a CEL formula \( \varphi \) we say that \( \mu \in \llbracket \text{MAX}(\varphi) \rrbracket(S, i, j) \) holds iff \( \mu \in \llbracket \varphi \rrbracket(S, i, j) \) and for all \( \mu' \in \llbracket \varphi \rrbracket(S, i, j) \), if \( \sup(\mu) \subseteq \sup(\mu') \), then \( \sup(\mu) = \sup(\mu') \) (i.e., \( \sup(\mu) \) is maximal with respect to set containment). Coming back to \( \rho_1 \), the \( \text{MAX} \) selector will output both \( \{1, 8\} \) and \( \{5, 8\} \), given that both complex events are maximal in terms of set inclusion. On the contrary, formula \( \rho_3 \) produced \( \{3, 6, 7\} \), \( \{3, 4, 7\} \), and \( \{3, 4, 6, 7\} \). Then, \( \text{MAX}(\rho_3) \) will only produce
\{3, 4, 6, 7\} as output, which is the maximal complex event. It is interesting to note that if we evaluate both \(\text{NXT}(\rho_3)\) and \(\text{LAST}(\rho_3)\) over the stream we will also get \{3, 4, 6, 7\} as the only output, illustrating that \(\text{NXT}\) and \(\text{LAST}\) also yield complex events with maximal information.

### 3.4.2. Compiling selection strategies into CEA

Selection strategies serve two purposes. On the one hand, they reduce the number of outputs, allowing users to focus on meaningful outputs. On the other hand, some selection strategies (e.g., \(\text{NEXT}\)) enable heuristics that allow for more efficient processing. Given these two purposes, two questions arise. First, what can we express with selection strategies that we cannot express with other operators? Second, do we need ad-hoc algorithms for evaluating selection strategies?

For the case of unary CEL, we answer these two questions negatively. Specifically, we show that we can compile any of the selection strategies into CEA. Because CEA are equivalent in expressive power to unary CEL by Theorem 2, it follows that selection strategies do not add expressive power to unary CEL. In addition, it follows that we can evaluate all selection strategies by compiling them into CEA and, therefore, we only need a single efficient algorithm for CEA evaluation (see Section 3.6).

We next discuss how to compile selection strategies into CEA. To this end, we extend our notation and allow selection strategies to be applied over CEA as follows. Given a CEA \(\mathcal{A}\), a selection strategy \(\text{SEL} \in \{\text{STRICT}, \text{NXT}, \text{LAST}, \text{MAX}\}\) and stream \(S\), the set of valuations \(\|\text{SEL}(\mathcal{A})\|_n(S)\) is defined analogously to \(\|\text{SEL}(\varphi)\|_n(S)\) for a formula \(\varphi\). Then, we say that a CEA \(\mathcal{A}_1\) is valuation equivalent to \(\text{SEL}(\mathcal{A}_2)\) if \(\|\mathcal{A}_1\|_n(S) = \|\text{SEL}(\mathcal{A}_2)\|_n(S)\) for every stream \(S\) and position \(n\).

**Theorem 3.** Let \(\text{SEL}\) be a selection strategy. For any CEA \(\mathcal{A}\), there is a CEA \(\mathcal{A}_{\text{SEL}}\) that is valuation equivalent to \(\text{SEL}(\mathcal{A})\). Furthermore, \(\mathcal{A}_{\text{SEL}}\) is I/O-deterministic and its size is at most exponential with respect to the size of \(\mathcal{A}\).

We note that the compilation of selection strategies reach two goals simultaneously: the resulting CEA computes the selection strategy and it is I/O-deterministic.
This last goal will be of special interest in Section 3.6, where we give an evaluation algorithm for I/O-deterministic CEA.

**Proof.** Let \( \mathcal{A} = (Q, \Delta, I, F) \) be a CEA and \( \mathcal{P} \) be the set of all predicates in the transitions of \( \Delta \). Moreover, let \( \mathcal{L} \) be the set of all variables sets used in \( \Delta \). For the next constructions, we extend the transition relation \( \Delta \) as a function:

\[
\Delta(R, P, L) = \{ q \in Q \mid \exists p \in R, P' \in \mathcal{P} : P \subseteq P' \land (p, P', L, q) \in \Delta \}
\]

for every \( R \subseteq Q, P \in \text{types}(\mathcal{P}), \) and \( L \in \mathcal{L} \). Further, we define \( \Delta(R, P) = \bigcup_{L \in \mathcal{L}} \Delta(R, P, L) \). We now give the construction of \( \mathcal{A}_{\text{SEL}} \) for each selector \( \text{SEL} \in \{\text{STRICT}, \text{NXT}, \text{LAST}, \text{MAX}\} \).

**Strict operator**

The principle behind the following construction is that a run that outputs an interval consists of three parts: the first and last parts that have only \( \emptyset \)-transitions and the middle part that has only non-\( \emptyset \)-transitions. Then, the construction is just a determinization of \( \mathcal{A} \) where each state also keeps track of the part of the run it is in.

We define \( \mathcal{A}_{\text{STRICT}} = (Q_{\text{STRICT}}, \Delta_{\text{STRICT}}, I_{\text{STRICT}}, F_{\text{STRICT}}) \) component by component. First, we define the set of states as \( Q_{\text{STRICT}} = 2^Q \times \{1, 2, 3\} \). Given \( i, i' \in \{1, 2, 3\} \) and \( L \in \mathcal{L} \), we say \( i, L, i' \) **preserve an interval** if either \( L \neq \emptyset \) and \( (i, i') \in \{(1, 2), (2, 2)\} \), or \( L = \emptyset \) and \( (i, i') \in \{(1, 1), (2, 3), (3, 3)\} \). Then, the set of transitions is defined as \( \Delta_{\text{STRICT}} = \{(R, i), P, L, (R', i') : i, L, i' \text{ preserve an interval and } R' = \Delta(R, P, L)\} \).

Finally, the sets of initial and final states are \( I_{\text{STRICT}} = \{(I, 1)\} \) and \( F_{\text{STRICT}} = \{(R, i) : R \cap F \neq \emptyset \text{ and } i \in \{1, 2, 3\}\} \).

This construction maintains the following invariant: for every run \( \rho \) of \( \mathcal{A} \) over \( S \) ending at some state \( q \), there is a run \( \rho' \) of \( \mathcal{A}_{\text{STRICT}} \) over \( S \) ending at some \( (R, i) \) with \( q \in R \) and \( \mu_\rho = \mu_\rho' \) if, and only if, \( \sup(\mu_\rho) \) is an interval. Moreover, one can see that \( \mathcal{A}_{\text{STRICT}} \) is I/O-deterministic, that its size is at most exponential over the size of \( \mathcal{A} \) and that, if \( \mathcal{A} \) is I/O-deterministic, then \( \mathcal{A}_{\text{STRICT}} \) is of size at most linear.

**Next operator**
We define $A_{\text{NXT}} = (Q_{\text{NXT}}, \Delta_{\text{NXT}}, I_{\text{NXT}}, F_{\text{NXT}})$ where $Q_{\text{NXT}} = 2^Q \times 2^Q$, $\Delta_{\text{NXT}}$ is as described below, $I_{\text{NXT}} = \{(I, \emptyset)\}$, and $F_{\text{NXT}} = \{(R, W) \mid R \cap F \neq \emptyset \land W \cap F = \emptyset\}$.

We say that a run $\rho_1$ of $A$ wins over a run $\rho_2$ of $A$ if $\sup(\mu_{\rho_2}) <_{\text{next}} \sup(\mu_{\rho_1})$. Intuitively, we will define the transition relation of $A_{\text{NXT}}$ in such a way that if a run $\rho$ of $A_{\text{NXT}}$ reaches state $(R, W)$ of $Q_{\text{NXT}}$, then $R$ represents a set of “current” states of $A$, i.e., those for which there is a run reaching them that has the same output as $\rho$. Importantly, $W$ will be the set of states of $A$ having runs that win over the current run $\rho$. As such, if $W$ contains a final state, then $\mu_\rho$ should not be output by $A_{\text{NXT}}$.

The behaviour above is reflected in the definition of the transition relation, which we define with the following two sets of transitions:

\[
\Delta_\emptyset = \{((R, W), P, \emptyset, (R', W')) \mid W' = \Delta(W, P) \land R' = \Delta(R, P, \emptyset) \setminus W'\},
\]

\[
\Delta_{\emptyset} = \{((R, W), P, \emptyset, (R', W')) \mid W' = \Delta(W, P) \cup \bigcup_{L \in L \setminus \{\emptyset\}} \Delta(R, P, L) \land R' = \Delta(R, P, \emptyset) \setminus W'\},
\]

where $P$ ranges over all predicates on $\text{types}(P)$ and $L$ over all variable sets on $L \setminus \{\emptyset\}$. Note that, from the extension of $\Delta$ to sets of states, it is implicitly required that $P \in \text{types}(P)$ (recall the definition and construction of $\text{types}(P)$ for the proof of Proposition 1). Then, $\Delta_{\text{NXT}} = \Delta_\emptyset \cup \Delta_{\emptyset}$. Basically, if the transition is marking, then the new set $W'$ of “winner” states extends $W$ using all possible transitions, while the new set of current states $R'$ extends $R$ using only marking transitions. Notice that if there is a state in both $W'$ and $R'$, it means there is a run that reaches the same state as the current run and has a better output, therefore making the current run useless according to the $\text{NXT}$ semantics; this is why we remove from $R'$ the states in $W'$. Conversely, if the transition is not marking, then the new set $W'$ extends $W$ with all possible transitions and also adds the states reachable from $R$ with marking transitions, while the new set $R'$ extends $R$ using only non-marking transitions.

The first thing we can notice about this construction is that $A_{\text{NXT}}$ is I/O-deterministic and that it is complete. Indeed, given a state $(R, S) \in Q_{\text{NXT}}$ and a mark $L$, for any tuple $t$ there is exactly one transition $((R, W), P, L, (R', W')) \in \Delta_{\text{NXT}}$ such that $t \in P$. Further, consider a stream $S = t_0t_1 \ldots$ and marks $L_0, L_1, \ldots, L_n$. Then, there is a run...
of $A_{\text{NXT}}$ over $S$:

$$\rho : (I, \emptyset) \xrightarrow{P_0, L_0} (R_1, W_1) \xrightarrow{P_1, L_1} \cdots \xrightarrow{P_n, L_n} (R_{n+1}, W_{n+1})$$

that satisfies the following invariants:

(i) $W_{n+1}$ is the set of all $q \in Q$ for which there is a run $\rho'$ of $A$ over $S$ at position $n$ ending at $q$ and $\sup(\mu_\rho) < \sup(\mu_{\rho'})$, and

(ii) $R_{n+1}$ is the set of all $q \in Q$ for which there is a run $\rho'$ of $A$ over $S$ at position $n$ ending at $q$ with $\mu_\rho = \mu_{\rho'}$, minus $W_{n+1}$.

These invariants guarantee that, whenever a run $\rho$ reaches some $(R, W)$ with $R \cap F \neq \emptyset$ and $W \cap F = \emptyset$, its output is $\prec_{\text{next}}$-maximal among the outputs of all accepting runs of $A$. Therefore, by setting this as our set of accepting states, it follows that $A_{\text{NXT}}$ is equivalent to $\text{NXT}(A)$.

Now, we analyse the size of the automaton $A_{\text{NXT}}$. First, one can see that $|Q_{\text{NXT}}|$ is bounded by $2^{|Q|^2}$, i.e., by the number of possible combinations of pairs of sets. Further, each state has $2 \cdot 2^{|\Delta|}$ possible transitions, two for each $P \in \text{types}(\mathcal{P})$, thus $|\Delta_{\text{NXT}}| \leq |Q_{\text{NXT}}| \cdot 2 \cdot 2^{|\Delta|}$. Therefore, the resulting automaton $A_{\text{NXT}}$ is of size at most exponential over the size of $A$.

**Last operator**

This case follows an approach similar to the previous ones but, in this case, each state of the constructed CEA now keeps three sets of states instead of two. We define $A_{\text{LAST}} = (Q_{\text{LAST}}, \Delta_{\text{LAST}}, I_{\text{LAST}}, F_{\text{LAST}})$ where $Q_{\text{LAST}} = 2^Q \times 2^Q \times 2^Q$, $I_{\text{LAST}} = (I, \emptyset, \emptyset)$ and $F_{\text{LAST}} = \{(R, W, T) \mid R \cap F \neq \emptyset \land W \cap F = \emptyset\}$. For the following, we say that a run $\rho_1$ wins over a run $\rho_2$ if $\sup(\mu_{\rho_2}) < \text{last} \sup(\mu_{\rho_1})$. If a run $\rho$ of $A_{\text{LAST}}$ reaches a state $(R, W, T) \in Q_{\text{LAST}}$, $R$ represents a set of current states of $A$, $W$ is a set of states of $A$ having runs that win over $\rho$, and $T$ is a set of states of $A$ having runs such that $\rho$ wins over them.

The above is shown in the following definition of the transition function which, again, we divide into two sets of transitions. The first one, $\Delta_{\emptyset}$, consists of all tuples $((R, W, T), P, L, (R', W', T'))$ such that $W' = \cup_{L \in L \setminus \{\emptyset\}} \Delta(W, P, L)$, $R' = \Delta(R, P, L)$.
$W'$ and $T' = (\Delta(T, P) \cup \Delta(R \cup W, P, \emptyset)) \setminus (R' \cup W')$. The second one, $\Delta_{\emptyset}$, consists of all tuples $((R, W, T), P, \emptyset, (R', W', T'))$ such that $W' = \Delta(W, P) \cup (\cup_{L \in \mathcal{C} \setminus \{\emptyset\}} \Delta(R \cup T, P, L))$, $R' = \Delta(R, P, \emptyset) \setminus W'$ and $T' = \Delta(T, P, \emptyset) \setminus (R' \cup W')$. Then, $\Delta_{\text{LAST}} = \Delta_{\emptyset} \cup \Delta_{\emptyset}$.

One can derive an explanation similar to the one in the $\text{NXT}$ case for why this construction maintains the sets of “winner”, “current” and “loser” states correctly. For this, we give a useful notion to keep in mind. Let $\rho_1, \rho_2$ be two runs of $\mathcal{A}$ that reach the same state $q \in Q$. Further, for $i \in \{1, 2\}$, let $L_i$ be the mark of the last transition of $\rho_i$ and let $\rho'_i$ be $\rho_i$ without its last transition. Now, let us consider the following tree of cases regarding $\sup(\mu_{\rho'_i})$ and $L_i$:

- If $\sup(\mu_{\rho'_1}) = \sup(\mu_{\rho'_2})$:
  - If $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$: then, $\sup(\mu_{\rho_1}) = \sup(\mu_{\rho_2})$;
  - If $L_1 = \emptyset$ and $L_2 \neq \emptyset$: then, $\sup(\mu_{\rho_1}) <_{\text{last}} \sup(\mu_{\rho_2})$;
  - If $L_1 \neq \emptyset$ and $L_2 = \emptyset$: then, $\sup(\mu_{\rho_2}) <_{\text{last}} \sup(\mu_{\rho_1})$.

- If $\sup(\mu_{\rho'_1}) <_{\text{last}} \sup(\mu_{\rho'_2})$:
  - If $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$: then, $\sup(\mu_{\rho_1}) <_{\text{last}} \sup(\mu_{\rho_2})$;
  - If $L_1 = \emptyset$ and $L_2 \neq \emptyset$: then, $\sup(\mu_{\rho_1}) <_{\text{last}} \sup(\mu_{\rho_2})$;
  - If $L_1 \neq \emptyset$ and $L_2 = \emptyset$: then, $\sup(\mu_{\rho_2}) <_{\text{last}} \sup(\mu_{\rho_1})$.

The case $\sup(\mu_{\rho'_1}) <_{\text{last}} \sup(\mu_{\rho'_2})$ is analogous to $\sup(\mu_{\rho'_1}) <_{\text{last}} \sup(\mu_{\rho'_2})$. This shows how all runs of $\mathcal{A}$ compete according to the $<_{\text{last}}$ order, and one can check that is the behaviour behind the construction of $\Delta_{\text{LAST}}$.

Like in the previous construction, here the resulting automaton is I/O-deterministic and complete. Moreover, for any stream $S = t_0t_1 \ldots$ and marks $L_0, L_1, \ldots L_n$, there is a run of $\mathcal{A}_{\text{LAST}}$ over $S$:

$$\rho : (I, \emptyset, \emptyset) \xrightarrow{P_0 \cdot L_0} (R_1, W_1, T_1) \xrightarrow{P_1 \cdot L_1} \ldots \xrightarrow{P_n \cdot L_n} (R_{n+1}, W_{n+1}, T_{n+1})$$

that satisfies the following invariants:

- (i) $W_{n+1}$ is the set of all $q \in Q$ for which there is a run $\rho'$ of $\mathcal{A}$ over $S$ at position $n$ ending at $q$ and $\sup(\mu_{\rho}) <_{\text{last}} \sup(\mu_{\rho'})$. 

(ii) $R_{n+1}$ is the set of all $q \in Q$ for which there is a run $\rho'$ of $A$ over $S$ at position $n$ ending at $q$ with $\mu_\rho = \mu_{\rho'}$, minus $W_{n+1}$, and

(iii) $T_{n+1}$ is the set of all $q \in Q$ for which there is a run $\rho'$ of $A$ over $S$ at position $n$ ending at $q$ with $\sup(\mu_{\rho'}) <_{\text{last}} \sup(\mu_\rho)$, minus $R_{n+1} \cup W_{n+1}$.

Then, whenever a run $\rho$ reaches some $(R, W, T)$ with $R \cap F \neq \emptyset$ and $W \cap F = \emptyset$, its output is $<_{\text{last}}$-maximal among the outputs of all accepting runs of $A$. By setting this as our accepting states, it follows that $A_{\text{LAST}}$ is equivalent to $\text{LAST}(A)$. Moreover, one can check that the size of $A_{\text{LAST}}$ is at most exponential over the size of $A$.

**Max operator**

For this case, we use a similar construction as for $\text{NXT}$, maintaining only two sets of states in each state of the resulting CEA. Formally, we define the automaton $A_{\text{MAX}} = (Q_{\text{MAX}}, \Delta_{\text{MAX}}, I_{\text{MAX}}, F_{\text{MAX}})$ with $Q_{\text{MAX}} = 2^Q \times 2^Q$, $I_{\text{MAX}} = \{(I, \emptyset)\}$ and $F_{\text{MAX}} = \{(R, W) \mid R \cap F \neq \emptyset \land W \cap F = \emptyset\}$. Again, for each $(R, W) \in Q_{\text{MAX}}$, $R$ and $W$ are the set of current and winner nodes, respectively, using the subset operation $\subset$ as the notion of “winning”. The transition relation in this case is defined by:

$$\Delta_{\emptyset} = \{((R, W), P, L, (R', W')) \mid W' = \bigcup_{L \in L \setminus \{\emptyset\}} \Delta(W, P, L) \land R' = \Delta(R, P, L) \setminus W'\}$$

$$\Delta_{\emptyset} = \{((R, W), P, \emptyset, (R', W')) \mid W' = \Delta(W, P) \cup \bigcup_{L \in L \setminus \{\emptyset\}} \Delta(R, P, L) \land R' = \Delta(R, P, \emptyset) \setminus W'\}$$

where $P$ ranges over all predicates on $\text{types}(P)$ and $L$ over all variable sets on $L \setminus \{\emptyset\}$. The resulting automaton is I/O-deterministic and, further, satisfies the same invariants of the $\text{NXT}$ construction, replacing $<_{\text{next}}$ with $\subset$. The size of the construction is at most exponential over the size of $A$. $\Box$
3.5. Expressiveness of Fragments of CEL

In this section, we seek to get a deeper understanding of the relationship in expressive power between non-contiguous sequencing and iteration; contiguous sequencing and iteration; and selection strategies.

In Section 3.5.1 we study the expressive power of CEL (and hence, non-contiguous sequencing and iteration) by identifying a fragment of CEA that coincides with unary CEL. One implication of this study is that, as expected, contiguous sequencing ($\colon$), contiguous iteration ($\oplus$) and the strict selection strategy STRICT are not expressible in CEL. Then, in Section 3.5.2, we study the expressiveness of CEL extended with different contiguous operators from $\{\colon, \oplus, \text{STRICT}\}$.

3.5.1. Unary CEL and the $\ast$ property

We first observe the following property of CEL formulas $\varphi$, whose proof is by induction on $\varphi$.

**Lemma 2.** For every CEL formula $\varphi$, if $\mu \in \llbracket \varphi \rrbracket (S, i, j)$, then

(i) $j \in \sup(\mu)$, and

(ii) $\emptyset \neq \sup(\mu) \subseteq \{i, \ldots, j\}$.

The formal semantics of a CEL formula clearly depends on the positions $i$ and $j$ that indicate where we start and end evaluation. In the following lemma, we observe that CEL is actually relatively insensitive to when we start evaluating, in two ways. The proof is by induction on $\varphi$.

**Lemma 3.** For every CEL formula $\varphi$, every stream $S$, all positions $i \leq j$, and all valuations $\mu$ it holds that, if $\mu \in \llbracket \varphi \rrbracket (S, i, j)$ then also

(i) $\mu \in \llbracket \varphi \rrbracket (S, \min(\sup(\mu)), \max(\sup(\mu)))$; and

(ii) $\mu \in \llbracket \varphi \rrbracket (S, i', j)$, for every $i' \leq i$.

The $\ast$-property. We next introduce the $\ast$-property of stream functions. Intuitively, this property states that the data outside a matched complex event $C$ does not effect whether or not $C$ is matched by a pattern.
For every stream $S$ and complex event, let $S[C]$ refer to the subsequence of $S$ induced by $C$, i.e.,

$$S[C] = S[i_1]S[i_2] \ldots S[i_k]$$

where $C = \{i_1, \ldots, i_k\}$ with $i_1 < i_2 < \cdots < i_k$. A stream-function is a function $f : \text{streams}(\mathcal{R}) \times \mathbb{N} \to 2^C$, where $\text{streams}(\mathcal{R})$ is the set of all $\mathcal{R}$-streams and $C$ is the set of all complex events. A stream-function hence specifies, for every stream $S$ and position $n$, the set of complex events $f(S, n)$ that it will output when it has observed the prefix $S[0], S[1], \ldots, S[n]$ of $S$. We require in particular that complex events that are output at position $n$ can only describe the prefix seen so far, i.e. that $f(S, n) \subseteq 2^{\{0,\ldots,n\}}$, for every $S$ and $n$. Although $f$ can in principle be any function that returns a set of complex events on pairs $(S, n)$, we are interested in the stream-functions $f$ that can be described either by a CEL formula $\varphi$ (i.e. $f(S, n) = \lceil \varphi \rceil_n(S)$) or by a CEA $A$ (i.e. $f(S, n) = \lceil A \rceil_n(S)$). Let $S_1$, $S_2$ be two streams and $C_1$, $C_2$ be two complex events. We say that $S_1$ and $C_1$ are ∗-related with $S_2$ and $C_2$, written as $(S_1, C_1) =_\ast (S_2, C_2)$, if $S_1[C_1] = S_2[C_2]$.

**Definition 8.** A stream-function $f$ has the ∗-property if the following hold, for every stream $S$, every position $n$ and every complex event $C$:

- Whenever $f$ outputs a complex event at position $n$, that position is also in the complex event, i.e., if $C \in f(S, n)$ then $n \in C$;
- If $C \in f(S, n)$ then also $C' \in f(S', \max(C'))$ for every $S'$ and $C'$ such that $(S, C) =_\ast (S', C')$.

A way to understand the ∗-property is to see $S'$ as the result of fixing the tuples in $S$ that are part of $S[C']$ and adding or removing other tuples arbitrarily, and defining $C'$ to be the complex event that has the same original tuples of $C$. Because of the requirement in the first bullet, $C'$ can only be output at position $\max(C')$ in $f$. Note that $C$ is necessarily non-empty (because of the first bullet) and, hence, because $(S, C) =_\ast (S', C')$ also $C'$ must be non-empty. As such, $\max(C')$ is always well-defined.

The following proposition shows that that CEL has the ∗-property.
**Proposition 2.** The function that maps $(S, n) \mapsto \llbracket \varphi \rrbracket_n(S)$ has the $*$-property, for every (not necessarily unary) CEL formula $\varphi$.

**Proof.** Let $\varphi$ be a CEL formula, let $S$ be a stream, let $n$ be a position and $C$ a complex event. Lemma 2 already ensures that if $C \in \llbracket \varphi \rrbracket_n(S)$ then $n \in C$. It hence suffices to show that

$$\text{if } C \in \llbracket \varphi \rrbracket_n(S) \text{ and } (S, C) =_* (S', C') \text{ then also } C' \in \llbracket \varphi \rrbracket_{\text{max}(C')}(S'). \quad (3.2)$$

To that end, we prove a stronger statement, which requires a number of auxiliary definitions. First, define a *correspondence* to be any function $\alpha$ that maps positions in a finite set $\text{dom}(\alpha) \subseteq \mathbb{N}$ to other positions in $\mathbb{N}$ such that $\alpha$ is strictly increasing, i.e., for any $i, j \in \text{dom}(\alpha)$ with $i < j$ we have $\alpha(i) < \alpha(j)$. We extend correspondences point-wise to complex events and valuations, i.e., if $C$ is a complex event with $C \subseteq \text{dom}(\alpha)$ then $\alpha(C) = \{ \alpha(i) \mid i \in C \}$ and, if $\mu$ is a valuation with $\sup(\mu) \subseteq \text{dom}(\alpha)$, then $\alpha(\mu)$ is the valuation such that $\alpha(\mu)(A) = \alpha(\mu(A))$, for every variable $A$. Let us say that stream $S_1$ is $\alpha$-related to stream $S_2$, denoted $S_1 =_\alpha S_2$ if $S_1[i] = S_2[\alpha(i)]$, for every position $i \in \text{dom}(\alpha)$. We will prove that for all $S_1, S_2, i, j, \mu$ and all correspondences $\alpha$ with $\{i, j\} \cup \sup(\mu) \subseteq \text{dom}(\alpha)$ we have:

$$\text{if } \mu \in \llbracket \varphi \rrbracket(S_1, i, j) \text{ and } S_1 =_\alpha S_2 \text{ then also } \alpha(\mu) \in \llbracket \varphi \rrbracket(S_2, \alpha(i), \alpha(j)). \quad (3.3)$$

From this (3.2) follows. Indeed, assume that $(S, C) =_* (S', C')$. If $C \in \llbracket \varphi \rrbracket_n(S)$ then there exists a valuation $\mu$ such that $C = \sup(\mu)$ and $\mu \in \llbracket \varphi \rrbracket(S, 0, n)$. By Lemma 2, $C$ is non-empty, $C \subseteq \{0, \ldots, n\}$, and $n \in C$. In particular, $n = \max(C)$. Moreover, by Lemma 3, also $C \in \llbracket \varphi \rrbracket(S, \min(C), n)$. Because $(S, C) =_* (S', C')$ we also know $C$ and $C'$ have the same size. Assume that $C = \{i_1, \ldots, i_k\}$ and $C' = \{i'_1, \ldots, i'_k\}$ with $\min(C) = i_1 < i_2 < \cdots < i_k = n = \max(C)$ and $\min(C') = i'_1 < i'_2 < \cdots < i'_k = \max(C')$. Let $\alpha$ be the correspondence that maps $i_m \mapsto i'_m$ for all $m \in [1, k]$. Then, because $S_1[C_1] = S_2[C_2]$ clearly, $S_1 =_\alpha S_2$ and $\alpha(C) = C'$. Moreover, because $\alpha$ is strictly increasing, $\max(C') = \max(\alpha(C)) = \alpha(\max(C)) = \alpha(n)$. Because $\mu \in \llbracket \varphi \rrbracket(S, \min(C), \max(C))$ we know by (3.3) that $\alpha(\mu) \in \llbracket \varphi \rrbracket(S_2, \alpha(\min(C)), \alpha(\max(C)))$ and therefore $C' = \sup(\alpha(\mu)) \in \llbracket \varphi \rrbracket_{\max(C')}(S')$.
It follows that \( \{ \varphi \} (S_2, \alpha(\min(C)), \alpha(\max(C))) \). Then, by Lemma 3, \( C' \in [\varphi](S_2, 0, \alpha(\max(C))) \). Hence, \( C'' \in [\varphi]_{\max(C')}(S_2) \), as desired.

The proof of property (3.3) is by induction on \( \varphi \). We only show the following illustrative case. Fix \( S_1, S_2, i, j, \mu, \alpha \) and assume that \( \{ i, j \} \cup \sup(\mu) \subseteq \text{dom}(\alpha), \mu \in [\varphi](S_1, i, j) \), and \( S_1 =_n S_2 \).

**Case** \( \varphi = \phi_1 ; \phi_2 \).

Then there exists \( k \in \mathbb{N} \) and valuations \( \mu_1 \) and \( \mu_2 \) such that \( \mu = \mu_1 \cup \mu_2, \mu_1 \in [\phi_1](S_1, i, k) \) and \( \mu_2 \in [\phi_2](S_1, k+1, j) \). By Lemma 2, \( k \in \text{sup}(\mu_1) \subseteq \text{sup}(\mu) \). It follows that \( \{ i, k \} \cup \text{sup}(\mu_1) \subseteq \{ i, j \} \cup \text{sup}(\mu) \subseteq \text{dom}(\alpha) \) and we may hence apply the induction hypothesis on \( \phi_1 \) to obtain that \( \alpha(\mu_1) \in [\phi_1](S_2, \alpha(i), \alpha(k)) \). To apply the induction hypothesis on \( \phi_2 \) we need a bit more work. First, from \( \mu_2 \in [\phi_2](S_1, k+1, j) \) and Lemma 3 we obtain that also \( \mu_2 \in [\phi_2](S_1, \text{min}(\text{sup}(\mu_2)), j) \). Clearly, \( \sup(\mu_2) \subseteq \text{sup}(\mu) \subseteq \text{dom}(\alpha) \). It follows that \( \{ \text{min}(\text{sup}(\mu_2)), j \} \cup \text{sup}(\mu_2) \subseteq \text{dom}(\alpha) \) and we may hence apply the induction hypothesis on \( \phi_2 \) to obtain that \( \alpha(\mu_2) \in [\phi_2](S_2, \alpha(\text{min}(\text{sup}(\mu_2))), \alpha(j)) \). We next wish to show that this implies that \( \alpha(\mu_2) \in [\phi_2](S_2, \alpha(k) + 1, \alpha(j)) \) because then \( \alpha(\mu_1) \cup \alpha(\mu_2) \in [\phi_1 ; \phi_2](S_2, \alpha(i), \alpha(j)) \), which is our desired result since \( \alpha(\mu) = \alpha(\mu_1 \cup \mu_2) = \alpha(\mu_1) \cup \alpha(\mu_2) \). We may derive that \( \alpha(\mu_2) \in [\phi_2](S_2, \alpha(k) + 1, \alpha(j)) \) from \( \alpha(\mu_2) \in [\phi_2](S_2, \alpha(\text{min}(\text{sup}(\mu_2))), \alpha(j)) \) using Lemma 3, provided that \( \alpha(k) + 1 \leq \alpha(\text{min}(\text{sup}(\mu_2))) \). To see that \( \alpha(k) + 1 \leq \alpha(\text{min}(\text{sup}(\mu_2))) \), we reason as follows. First, observe that because \( \mu_2 \in [\phi_2](S_1, k+1, j) \) we know from Lemma 2 that \( \text{sup}(\mu_2) \subseteq \{ k + 1, \ldots, j \} \); therefore \( k + 1 \leq \text{min}(\text{sup}(\mu_2)) \). Hence, \( k < k+1 \leq \text{min}(\text{sup}(\mu_2)) \). Then, because \( \alpha \) is strictly increasing, we know that \( \alpha(k) < \alpha(\text{min}(\text{sup}(\mu_2))) \). Therefore, \( \alpha(k) + 1 \leq \alpha(\text{min}(\text{sup}(\mu_2))) \).

□

We note that the CEL operators in \{\( ;, \oplus, \pi_L, \text{START, UNLESS} \)}, as well as the selection operators \text{NXT, STRICT, LAST, and MAX} allow to express stream functions that do not have the *-property. These operators are therefore not expressible in CEL. To see this for the case of \text{UNLESS}, consider \( \varphi = R \text{ UNLESS } T \). Then \( [\varphi]_n(S) \) consists of at most one complex event, namely the complex event \( C = \{ n \} \) such
that \( S[n] \in \text{tuples}(R) \) and there is no \( i < n \) with \( S[i] \in \text{tuples}(T) \). Now consider the stream \( S_1 = t_0t_1 \ldots \) such that both \( t_0 \) and \( t_1 \) are in \( \text{tuples}(R) \). Take \( S_2 \) equal to \( S_1 \) but change the first event \( t_0 \) to some event \( t'_0 \) of type \( T \). Then clearly, \( \{1\} \in [\varphi]_1(S_1), \{1\} \not\in [\varphi]_1(S_2) \) and yet \((S_1, \{1\}) \equiv_r (S_2, \{1\})\). Therefore, \( \varphi \) does not have the \( * \)-property, and is hence not expressible in unary CEL. Similar patterns can be constructed for the other operators.

**Corollary 2.** The CEL operators in \( \{:, \oplus, \pi_L, \text{START}, \text{UNLESS}\} \), as well as the selection operators \( \{\text{NXT}, \text{STRICT}, \text{LAST}, \text{MAX}\} \) are not expressible in CEL.

By contrast, AND and ALL are expressible in unary CEL, as we will see next.

We already know that every unary CEL formula can be translated into a valuation-equivalent CEA (Theorem 1). Note that this CEA necessarily has the \(*\)-property. We will next show that the converse also holds: every CEA that has the \(*\)-property can be translated into an equivalent unary CEL formula. Towards this goal, we first introduce a new semantics on CEA, called the \(*\)-semantics.

**Definition 9.** Let \( A = (Q, \Delta, I, F) \) be a complex event automaton and \( S = t_1t_2 \ldots \) be a stream. A \(*\)-run of \( A \) over \( S \) ending at \( n \) is a sequence of transitions:

\[ \rho^* : (q_0, 0) \xrightarrow{P_1/L_1} (q_1, i_1) \xrightarrow{P_2/L_2} \ldots \xrightarrow{P_k/L_k} (q_k, i_k) \]

such that \( q_0 \in I, 0 < i_1 < \ldots < i_k = n \) and, for every \( j \geq 1 \), \((q_{j-1}, P_j, L_j, q_j) \in \Delta, L_j \neq \emptyset \) and \( S[i_j] \in P_j \). We say that \( \rho^* \) is an accepting \(*\)-run if \( q_k \in F \). Furthermore, we denote the valuation induced by \( \rho^* \) as \( \mu_{\rho^*} \) such that for every variable \( A \), \( \mu_{\rho^*}(A) = \{i_k \mid A \in L_i\} \). The set of all valuations generated by \( A \) over \( S \) under the \(*\)-semantics is defined as: \( \|A\|^*_n(S) = \{\mu_{\rho^*} \mid \rho^* \text{ is an accepting \(*\)-run of } A \text{ over } S \text{ ending at } n\} \). The set of all complex events generated by \( A \) over \( S \) under the \(*\)-semantics is then \( \|A\|^*_n(S) = \{\sup(\mu) \mid \mu \in \|A\|^*_n(S)\} \).

Notice that under this semantics, the automaton no longer has the ability to inspect a tuple without marking it but it is allowed to skip an arbitrary number of tuples between two marking transitions. The following proposition states the relation that exists between the \(*\)-property and the \(*\)-semantics over CEA.

**Proposition 3.** If the stream-function defined by a CEA \( A \) has the \(*\)-property, then \( \|A\|^*_n(S) = [\|A\|^*_n(S) \text{ for every } S \text{ and } n. \)
Proof. Consider any CEA $\mathcal{A} = (Q, \Delta, I, F)$ that has the $*$-property. We prove that $[\mathcal{A}]^*_n(S) = [\mathcal{A}]^*_n(S)$. First, consider a complex event $C \in [\mathcal{A}]^*_n(S)$. This means that there is an accepting run $\rho$ of $\mathcal{A}$ of the following form such that $\sup(\mu_\rho) = C$,

$$\rho : q_0 \xrightarrow{P_1/L_1} q_1 \xrightarrow{P_2/L_2} \cdots \xrightarrow{P_n/L_n} q_n.$$

By definition of the $*$-property, position $n \in C$. In particular, $C$ is non-empty. Assume that $C = \{i_1, i_2, \ldots, i_k\}$ with $k \geq 1$, and consider the stream $S'$ whose first $k$ elements are the events $S[i_1]S[i_2]\ldots S[i_k]$ and whose subsequent elements are arbitrary (but fixed). Let $C' = \{1, \ldots, k\}$. Clearly, $(S, C) =_\ast (S', C')$ and $\max(C') = k$. Then, because $\mathcal{A}$ defines a stream-function with $*$-property, we know that $C' \in [\mathcal{A}]^*_k(S')$. There hence has to be an accepting run of length $k$ of $\mathcal{A}$ over $S'$ of the form:

$$\rho' : q_0' \xrightarrow{P_1'/L_1'} q_1' \xrightarrow{P_2'/L_2'} \cdots \xrightarrow{P_k'/L_k'} q'_k,$$

with $\sup(\mu_{\rho'}) = C'$ and, therefore, each $L_i' \neq \emptyset$. By definition this yields the following accepting $*$-run of $\mathcal{A}$ over $S'$:

$$\sigma' : (q_0', 0) \xrightarrow{P_1'/L_1'} (q_1', 1) \xrightarrow{P_2'/L_2'} \cdots \xrightarrow{P_k'/L_k'} (q'_k, k).$$

Then, because $S'[C'] = S[C]$ the following is a valid accepting $*$-run of $\mathcal{A}$ over $S$.

$$\sigma : (q_0', 0) \xrightarrow{P_1'/L_1'} (q_1', i_1) \xrightarrow{P_2'/L_2'} \cdots \xrightarrow{P_k'/L_k'} (q'_k, i_k).$$

Therefore, $\sup(\mu_\sigma) = C \in [\mathcal{A}]^*_n(S)$.

The proof for the converse case is similar. Assume that $C \in [\mathcal{A}]^*_n(S)$, which means that the $*$-run $\sigma$ of $\mathcal{A}$ over $S$ exists. Because $S[C] = S'[C']$, the $*$-run $\sigma'$ of $\mathcal{A}$ over $S'$ also exists, which must coincide with a normal run $\rho'$ of $\mathcal{A}$ over $S'$. Because $\mathcal{A}$ defines a function with $*$-property, the accepting run $\rho$ of $\mathcal{A}$ over $S$ has to exist. \hfill \Box

We can now effectively capture the expressiveness of unary CEL formulas at a complex event level as follows.

**Theorem 4.** At a complex event level, unary CEL has the same expressive power as CEA under the $*$-semantics. Namely, for every unary CEL formula $\varphi$ there exists a CEA $\mathcal{A}$ such that $[\varphi]^*_n(S) = [\mathcal{A}]^*_n(S)$ for every $S$ and $n$, and vice versa.
**Proof.** Let \( \varphi \) be a unary CEL formula. By Theorem 1 there exists a CEA \( \mathcal{A} \) such that \([\varphi]_n(S) = [\mathcal{A}]_n(S)\) for every \( S \) and \( n \). Since \([\varphi]\) has the \(*\)-property, so does \([\mathcal{A}]\).

Then \([\varphi]_n(S) = [\mathcal{A}]_n^*(S) = [\mathcal{A}]_n^*(S)\) for every \( S \) and \( n \) by Proposition 3.

Conversely, let \( \mathcal{A} = (Q, \Delta, I, F) \) be a CEA. We define a unary CEL formula \( \varphi_\mathcal{A} \) such that \([\mathcal{A}]_n^*(S) = [\varphi_\mathcal{A}]_n(S)\) for every stream \( S \) and \( n \in \mathbb{N} \).

Assume that \( Q = \{q_1, q_2, \ldots, q_n\} \). To simplify the construction, further assume that \( I = \{q_1\} \) and \( F = \{q_n\} \). This is without loss of generality: we can always ensure a single initial state by adding a new initial state and copying each out-transition of the original initial states as an out-transition of the new initial state; likewise, we can ensure a single final state by adding a new final state and copying all in-transitions of the original final states as an in-transition of the new final state. Define the formula \( \text{FALSE} \) as a formula that is never satisfied. One way to define it is \( \text{FALSE} = (\mathcal{R} \text{ FILTER } \emptyset) \).

Like the proof of Theorem 2, the main idea is based on the construction used to convert standard FSA into regular expressions. We define, for every pair of states \( q_i, q_j \), a CEL formula \( \varphi_{ij} \) that represents the complex events defined by the \(*\)-runs from \( q_i \) to \( q_j \). To aid in the definition of \( \varphi_{ij} \) we will first define, for every \( 1 \leq k \leq |Q| \) the formula \( \varphi_{ij}^k \) that represents the complex events defined by the \(*\)-runs from \( q_i \) to \( q_j \) that only visit states in \( \{q_1, \ldots, q_k\} \). It is then clear that \( \varphi_{ij}^{|Q|} = \varphi_{ij} \).

We define \( \varphi_{ij}^k \) recursively as follows. In the base case \( k = 0 \), for each \( i, j \), if there is no transition from \( q_i \) to \( q_j \) in \( \mathcal{A} \), then \( \varphi_{ij}^0 = \text{FALSE} \); otherwise we define it as:

\[
\varphi_{ij}^0 = \rho_{P_1} \lor \rho_{P_2} \lor \ldots \lor \rho_{P_k}
\]

where \( P_1, \ldots, P_k \) are all the predicates of the marking transitions from \( q_i \) to \( q_j \). Moreover, \( \rho_P \) represents the CEL formula that accepts all complex events that consist of a single event that satisfies \( P \). Concretely, assuming that the schema \( \mathcal{R} = \{R_1, \ldots, R_r\} \) we can define it as \( \rho_P := (R_1 \lor \cdots \lor R_r) \text{ FILTER } P' \) where \( P' \) is the second order extension of CEA predicate \( P \). Next, the recursion is defined as:

\[
\varphi_{ij}^k = \varphi_{ij}^{k-1} \lor (\varphi_{ik}^{k-1} \lor \varphi_{kj}^{k-1}) \lor (\varphi_{ik}^{k-1} \lor \varphi_{kk}^{k-1} \lor \varphi_{kj}^{k-1})
\]
Finally, the final formula $\varphi_A$ is the result of considering $\varphi_{1n}$. The correctness of the construction can be proved by doing induction over the number of states. □

Notice that Theorem 4 cannot be applied at a valuation level, mainly because CEA has the ability to rename variables, while CEL does not. For instance, consider the complex event automaton $A = (\{p, q\}, \{(p, (\text{tuples}(R), T), q)\}, \{p\}, \{q\})$ with $T \notin R$, which for all $n$ and $S$ defines $\|A\|^*_{n}(S) = \{\mu \mid \mu(T) = \{n\} \land S[n] \in \text{tuples}(R)\}$. We claim that there is no unary CEL formula $\varphi$ such that $\|\varphi\|^*_{n}(S) = \|A\|^*_{n}(S)$. The closest we can get is the formula $\varphi = R \text{ IN } T$, which recognizes $\|\varphi\|^*_{n}(S) = \{\mu \mid \mu(T) = \mu(R) = n \land S[n] \in \text{tuples}(R)\}$. Note that the valuations of $A$ do not define a value for $R$, while the ones of $\varphi$ do.

By combining Theorem 4 and Proposition 3 we get the following result.

**Corollary 3.** Let $f$ be a stream-function. Then, $f$ can be defined by a CEA and has the $*$-property if, and only if, there exists a unary CEL formula $\varphi$ such that $f(S,n) = [\varphi]_{n}(S)$, for every $S$ and $n$.

We can now use this characterization to prove that $\text{AND}$ and $\text{ALL}$ are expressible in unary CEL.

**Corollary 4.** For every expression $\varphi$ of the form $\varphi_1 \text{ OP } \varphi_2$, with $\text{OP} \in \{\text{AND}, \text{ALL}\}$ and $\varphi_1, \varphi_2$ unary CEL formulae, there is a unary CEL formula $\varphi'$ such that $[\varphi']_{n}(S) = [\varphi]_{n}(S) \land [\varphi_1]_{n}(S)$ for every $S$ and $n$.

**Proof.** We illustrate the proof idea for $\varphi = \varphi_1 \text{ AND } \varphi_2$. The idea for $\varphi = \varphi_1 \text{ ALL } \varphi_2$ is similar.

The proof is by application of Corollary 3, for which we need to show that (1) $[\varphi]$ has the $*$-property and (2) can be defined by means of a CEA. Showing that it has the $*$-property can be done analogously to the proof of Proposition 2, namely by showing the stronger property (3.3) introduced in that proof and using the fact that this stronger property is already proved for $\varphi_1$ and $\varphi_2$ in the Proof of Proposition 2. That $\text{AND}$ can be defined by means of a CEA follows from Theorem 1. □
3.5.2. Strict Sequencing versus Strict Selection

There exists a relation between the \textsc{strict} selection strategy and the strict sequencing (\( : \)) and strict iteration operators (\( \oplus \)): some formulas using the former can be expressed with the latter and vice-versa. For example, formulas \( R : S \) and \( R \oplus \) have (complex-event) equivalent formulas \( \textsc{strict}(R ; S) \) and \( \textsc{strict}(R+) \), respectively.

In the following, we show that this relation depends on the set of predicates allowed but, in general, having \( : \) and \( \oplus \) together provides more expressiveness than having \textsc{strict}. Because the proofs of this subsection are technical and rather long, we defer them to Appendix to avoid disrupting the flow of the discussion.

Recall that, given a set of operators \( O \), we define \( \text{CEL} \cup O \) to be the fragment of CEL formulas that can be defined by using operators of CEL plus the operators in \( O \). For instance, \( \text{strict}(R ; T)_+ \) is in \( \text{CEL} \cup \{\text{strict}\} \), but \( (R : T)_+ \) is not. Moreover, to compare the expressiveness of two fragments with operators \( O \) and \( O' \), we say that fragment \( \text{CEL} \cup O \) is contained in fragment \( \text{CEL} \cup O' \), and use notation \( \text{CEL} \cup O \subseteq \text{CEL} \cup O' \), to state that for every formula in \( \text{CEL} \cup O \) there exists a complex-event equivalent formula in \( \text{CEL} \cup O' \). We use \( \text{CEL} \cup O \subset \text{CEL} \cup O' \) to denote that \( \text{CEL} \cup O \subseteq \text{CEL} \cup O' \) and \( \text{CEL} \cup O' \) is strictly more expressive: there exists a formula in \( \text{CEL} \cup O' \) for which no complex-event equivalent formula exists in \( \text{CEL} \cup O \).

Lastly, to compare two fragments we often add the restriction that both use only certain predicates. Given a fragment \( \text{CEL} \cup O \) and a set of predicates \( \mathcal{P} \), we denote by \( \text{CEL} \cup O(\mathcal{P}) \) the set of formulas in \( \text{CEL} \cup O \) that only use predicates of \( \mathcal{P} \) in their filters. The expressiveness relation between fragments with different contiguous operators is summarized in Fig. 3.6.

As mentioned earlier, the formula \( R : T \) in \( \text{CEL} \cup \{\cdot\} \) has an equivalent formula \( \text{strict}(R ; T) \) in \( \text{CEL} \cup \{\text{strict}\} \). In a similar way, a more complex formula like \( R ; T + : U \) also has an equivalent formula, namely: \( R ; (\text{strict}(T ; U) \text{ or } T + \text{strict}(T ; U)) \). This kind of transformation can be extended to all formulas in \( \text{CEL} \cup \{\cdot\} \).

The expressive containment is strict: in \( \text{CEL} \cup \{\text{strict}\} \) the combination \( \text{strict} \) along with \( + \) allows to verify an unbounded number of strict-concatenation of events
Figure 3.6. Expressiveness comparison between fragments with different contiguous operators. By Proposition 4, containment (a) holds and is strict for any set of predicates \( \cal P \). By Proposition 5, (c) holds for any set of predicates \( \cal P \). By Theorem 5, (b) holds whenever \( \cal P \) contains a binary equality predicate in which case (c) becomes strict. By Proposition 6, (c) becomes \( \equiv \) whenever \( \cal P \) is a set of unary predicates.

in the complex event, even allowing to simulate \( \oplus \) in some cases, while in \( \text{CEL} \cup \{ : \} \) each : verifies at most one strict concatenation. For instance, formula \( \text{STRICT}(R^+) \) has no equivalent formula in \( \text{CEL} \cup \{ : \} \).

**Proposition 4.** For any set of CEL predicates \( \cal P \), it holds that \( \text{CEL} \cup \{ : \}(\cal P) \subset \text{CEL} \cup \{ \text{STRICT} \}(\cal P) \).

**Proof.** For this proof, we show by induction that for any formula of the form \( \varphi_1 : \varphi_2 \) we can remove the : operator using STRICT instead. For the induction, we consider all possible cases of \( \varphi_1 \). The case when \( \varphi_1 = \psi \text{ IN } A \) for some \( \psi \) complicates the proof, which is why we begin by showing how to get rid of this case, to then proceed with the inductive proof. For this strategy, the following notation will be useful: for any formula \( \rho \) and variables \( A, B \) we write \( \rho 
 A \rightarrow B \) to refer to the formula \( \rho \) after replacing every occurrence of \( A \) by \( B \).

As discussed above, we start by removing the \( \text{IN} \) operator. Specifically, we show that any formula in \( \text{CEL} \cup \{ : \}(\cal P) \) can be rewritten as an equivalent formula such that every subformula \( \psi \text{ IN } A \) has the form \( \psi = (R \text{ IN } A_1 \ldots \text{ IN } A_k) \) for some \( R, A_1, \ldots, A_k \). In other words, we can push the \( \text{IN} \) operators to the “atomic” level of the formula. This can be done by showing that any formula of the form \( \varphi = \varphi' \text{ IN } A \) can be rewritten as a new formula that has the operator \( \text{IN} \) applied one level lower than \( \varphi \). This is done in the following way:

- If \( \varphi' = \rho_1 \text{ OP } \rho_2 \) with \( \text{OP} \in \{ ; , : , \text{OR} \} \), then \( \varphi \equiv (\rho_1 \text{ IN } A) \text{ OP } (\rho_2 \text{ IN } A) \).
- If \( \varphi' = \rho \text{ OP } \) with \( \text{OP} \in \{ +, \oplus \} \), then \( \varphi \equiv (\rho \text{ IN } A) \text{ OP } \).
- If \( \varphi' = \rho \text{ FILTER } P(\vec{A}) \), then
- if $A$ is not in $\bar{A}$ then $\varphi \equiv (\rho \mathbin{\text{IN}} A) \mathbin{\text{FILTER}} P(\bar{A})$, and
- if $A$ is in $\bar{A}$ then $\varphi \equiv (\rho^{A\rightarrow\bar{A}'} \mathbin{\text{IN}} A) \mathbin{\text{FILTER}} P(\bar{A}')$, where $A'$ is a new fresh variable and $\bar{A}'$ is $\bar{A}$ replacing $A$ with $A'$.

By applying these equivalences recursively, one can push every IN operator to the lowest level of the formula.

Now we prove that, for every formula $\varphi$ that has the form $\varphi_1 : \varphi_2$ with $\varphi_1$ and $\varphi_2$ in $\text{CEL} \cup \{\text{STRICT}\}(P)$, there exists a formula $\psi$ in $\text{CEL} \cup \{\text{STRICT}\}(P)$ equivalent to $\varphi$. By the previous discussion, we can assume that every IN operator in $\varphi$ is applied at the lowest level. Then the proof follows by doing induction over the structure of $\varphi$. The base case is when $\varphi = (R \mathbin{\text{IN}} A_1 \ldots \mathbin{\text{IN}} A_j) : (T \mathbin{\text{IN}} B_1 \ldots \mathbin{\text{IN}} B_k)$ for some $R$, $T$, $A_1$, $\ldots$, $A_j$, $B_1$, $\ldots$, $B_k$. Clearly it is the case that $\varphi$ is equal to the formula $\psi = \text{STRICT}((R \mathbin{\text{IN}} A_1 \ldots \mathbin{\text{IN}} A_j) ; (T \mathbin{\text{IN}} B_1 \ldots \mathbin{\text{IN}} B_k))$. For the inductive step, we consider each case separately:

- If $\varphi_1 = \rho_1 \mathbin{\text{OR}} \rho_2$, then $\varphi \equiv (\rho_1 : \varphi_2) \mathbin{\text{OR}} (\rho_2 : \varphi_2)$ and $\rho_2 : \varphi_2$ is smaller than $\varphi_1 : \varphi_2$. By induction hypothesis, $(\rho_2 : \varphi_2)$ has an equivalent formula $\sigma$ in $\text{CEL} \cup \{\text{STRICT}\}(P)$. Thus, $\psi = \rho_1 ; \sigma$ is equivalent to $\varphi$.
- If $\varphi_1 = \rho \mathbin{\text{FILTER}} P(\bar{A})$, then $\varphi \equiv (\rho^{\bar{A}\rightarrow\bar{A}'} : \varphi_2) \mathbin{\text{FILTER}} P(\bar{A}')$, where $\bar{A}' = (A'_1, \ldots, A'_{k})$ is a tuple of new variables with the same arity as $\bar{A} = (A_1, \ldots, A_k)$. By induction hypothesis, $(\rho^{\bar{A}\rightarrow\bar{A}'} : \varphi_2)$ has an equivalent formula $\sigma$ in $\text{CEL} \cup \{\text{STRICT}\}(P)$. Thus, $\psi = \sigma \mathbin{\text{FILTER}} P(\bar{A}')$ is equivalent to $\varphi$. Note that, since we renamed the variables $\bar{A}$ with $\bar{A}'$ in $\rho$, then for any filter $P'(\bar{B})$ with some $A_i \in \bar{B}$ that is applied in a higher level, we must also add the filter $P(\bar{B}')$ where $\bar{B}'$ is $\bar{B}$ replacing $A_i$ with $A'_i$.
- If $\varphi_1 = \rho \mathbin{\text{OR}} \rho_2$, then $\varphi \equiv (\rho_1 : \varphi_2) \mathbin{\text{OR}} (\rho_2 : \varphi_2)$. By induction hypothesis, both $(\rho_1 : \varphi_2)$ and $(\rho_2 : \varphi_2)$ have equivalent formulas $\sigma_1$ and $\sigma_2$, respectively, in $\text{CEL} \cup \{\text{STRICT}\}(P)$. Thus, $\psi = \sigma_1 \mathbin{\text{OR}} \sigma_2$ is equivalent to $\varphi$.
Note that $\varphi_1 = \rho_1 \text{ IN } A$ was not considered, given that \text{ IN } is included in the base case, by the first part of this proof. Finally, we should also consider when $\varphi_1 = (R \text{ IN } A_1 \ldots \text{ IN } A_j)$. For this, we can apply the induction step over $\varphi_2$ but these cases are analogous to the previous ones.

The expressive containment is strict: in $\text{CEL} \cup \{\text{STRICT}\}$ the combination $\text{STRICT}$ along with $+$ allows to verify an unbounded number of strict-concatenation of events in the complex event, even allowing to simulate $\oplus$ in some cases, while in $\text{CEL} \cup \{:\}$ each $:$ verifies at most one strict concatenation. For instance, formula $\varphi_{\text{STR}+} = \text{STRICT}(R +)$ has no equivalent formula in $\text{CEL} \cup \{:\}$. We formalize this by defining a property that every formula $\varphi$ in $\text{CEL} \cup \{:\}$ must satisfy, which basically says that the number of strict-concatenations that $\varphi$ can verify is bounded by the number of $:$ operations it has.

Given a stream $S = t_0t_1 \ldots$, an $R$-tuple $g$, a valuation $\mu$ and a position $i$ with $\min(\sup(\mu)) \leq i < \max(\sup(\mu))$, we define the insertion of $g$ at $i$ inside $(S, \mu)$ as the pair $(S', \mu')$ where:

- $S'$ is result of adding $t$ at position $i$ of $S$, i.e., $= t_0t_1 \ldots t_{i-1}gt_i t_{i+1} \ldots$;
- $\mu'$ is the result of moving positions of $\mu$ accordingly, that is, for every variable $A$, $\mu'(A) = \{j \mid j \in \mu(A) \land j < i\} \cup \{j + 1 \mid j \in \mu(A) \land j \geq i\}$.

A CEL formula $\varphi$ is said to be strict-bounded if there exists some number $N \geq 2$ such that, for every stream $S$, tuple $g$, positions $p_1, p_2$ and valuation $\mu \in \models(\varphi)(S, p_1, p_2)$ with $|\sup(\mu)| \geq N$, there exists some $i < p_2$ such that $\mu' \in \models(\varphi)(S', p_1, p_2 + 1)$, where $(S', \mu')$ is the insertion of $t$ at $i$ inside $(S, \mu)$. Roughly speaking, if the formula $\varphi$ uses $N$ operations $:$ and the valuation contains more than $N$ positions, then there are some positions where the strict-concatenation cannot be checked and thus we can insert there a garbage tuple $g$.

The following lemma can be proved by doing structural induction over formula $\varphi$.

**Lemma 4.** Every formula $\varphi$ in $\text{CEL} \cup \{:\}$ is strict-bounded.
Now, we use Lemma 4 to prove that there is no formula in \( \text{CEL} \cup \{ : \} \) equivalent to \( \varphi_{\text{STR}^+} \). By contradiction, assume there exists such formula, call it \( \psi \). By Lemma 4, \( \psi \) is strict-bounded, so let \( N \) be the strict-bounded constant for \( \psi \). Now, consider the stream \( S = RR \ldots \) and define \( \mu \) such that \( \mu(R) = \{0, 1, 2, \ldots , N\} \). Clearly, \( |\sup(\mu)| \geq N \) and, at position \( N \), \( \mu \in \| \varphi_{\text{STR}^+} \| (S, 0, N) \) and so \( \mu \in \| \psi \| (S, 0, N) \). Now, consider the stream \( S \), a tuple \( t \notin \text{tuples}(R) \), positions \( 0, N \) and valuation \( \mu \). Because \( \psi \) is strict-bounded, there exists some \( i < N \) such that \( \mu' \in \| \varphi \| (S', 0, N + 1) \), where \( (S', \mu') \) is the insertion of \( t \) at \( i \) inside \( (S, \mu) \). But then, since we added \( t \) in the middle of \( S \) and swapped some positions of \( \mu' \), \( \sup(\mu') \) is no longer a contiguous interval, and therefore \( \mu' \notin \| \varphi_{\text{STR}^+} \| (S', 0, N + 1) \), which contradicts the fact that \( \varphi_{\text{STR}^+} \) and \( \psi \) are equivalent.

\[ \square \]

Conversely, operators \( : \) and \( \oplus \) combined are in general more expressive than \( \text{STR} \).

**Proposition 5.** For any set of \( \text{CEL} \) predicates \( \mathcal{P} \), it is always the case that

\[ \text{CEL} \cup \{ \text{STR} \} (\mathcal{P}) \subseteq \text{CEL} \cup \{ :, \oplus \} (\mathcal{P}) . \]

**Proof.** Consider a formula \( \varphi \) in \( \text{CEL} \cup \{ \text{STR} \} (\mathcal{P}) \). We first prove that for every \( \varphi' \) in \( \text{CEL} \cup \{ \text{STR} \} (\mathcal{P}) \) there is a formula \( \psi' \) in \( \text{CEL} \cup \{ :, \oplus \} (\mathcal{P}) \) that is equivalent to \( \text{STR} (\varphi') \), for which we do induction over the structure of \( \varphi' \). The base case is \( \varphi' = R \), which already satisfies the above considering \( \psi' = R \). For the inductive step, consider the following cases.

- If \( \varphi' = \rho \text{ IN } A \), then \( \text{STR} (\varphi') \equiv \text{STR} (\rho) \text{ IN } A \). By induction hypothesis, there is a formula \( \sigma \) in \( \text{CEL} \cup \{ :, \oplus \} (\mathcal{P}) \) equivalent to \( \text{STR} (\rho) \). Thus, \( \psi' = \sigma \text{ IN } A \) is equivalent to \( \text{STR} (\varphi') \).
- If \( \varphi' = \rho_1 ; \rho_2 \), then \( \text{STR} (\varphi') \equiv \text{STR} (\rho_1) : \text{STR} (\rho_2) \). Then, formulas \( \text{STR} (\rho_1) \) and \( \text{STR} (\rho_2) \) both have equivalent formulas \( \sigma_1 \) and \( \sigma_2 \), respectively, in \( \text{CEL} \cup \{ :, \oplus \} (\mathcal{P}) \). Thus, \( \psi' = \sigma_1 : \sigma_2 \) is equivalent to \( \text{STR} (\varphi') \).
• If \( \varphi' = \rho \) FILTER \( P \), then \( \text{STRICT}(\varphi') \equiv \text{STRICT}(\rho) \) FILTER \( P \). By induction hypothesis, \( \text{STRICT}(\rho) \) has an equivalent formula \( \sigma \) in \( \text{CEL} \cup \{ :, \oplus \}(P) \). Thus, \( \psi' = \sigma \) FILTER \( P \) is equivalent to \( \text{STRICT}(\varphi') \).

• If \( \varphi' = \rho_1 \) OR \( \rho_2 \), then \( \text{STRICT}(\varphi') \equiv \text{STRICT}(\rho_1) \) OR \( \text{STRICT}(\rho_2) \). Again, both \( \text{STRICT}(\rho_1) \) and \( \text{STRICT}(\rho_2) \) have equivalent formulas \( \sigma_1 \) and \( \sigma_2 \), respectively, in \( \text{CEL} \cup \{ :, \oplus \}(P) \). Thus, \( \psi' = \sigma_1 \) OR \( \sigma_2 \) is equivalent to \( \text{STRICT}(\varphi') \).

• If \( \varphi' = \rho + \), then \( \text{STRICT}(\varphi') \equiv \text{STRICT}(\rho) \oplus \). By induction hypothesis, \( \text{STRICT}(\rho) \) has an equivalent formula \( \sigma \) in \( \text{CEL} \cup \{ :, \oplus \}(P) \). Thus, \( \psi' = \sigma \oplus \) is equivalent to \( \text{STRICT}(\varphi') \).

It is left to replace every subformula \( \text{STRICT}(\varphi') \) of \( \varphi \) with its \( \text{CEL} \cup \{ :, \oplus \}(P) \) equivalent \( \psi' \), and the resulting formula will be in \( \text{CEL} \cup \{ :, \oplus \}(P) \) and will be equivalent to \( \varphi \). \( \Box \)

The most interesting result of this section is that fragments \( \text{CEL} \cup \{ \text{STRICT} \} \) and \( \text{CEL} \cup \{ \oplus \} \) are incomparable. Clearly, \( \text{CEL} \cup \{ \text{STRICT} \} \not\subseteq \text{CEL} \cup \{ \oplus \} \) since formula \( \text{STRICT}(R; T) \) has no equivalent in \( \text{CEL} \cup \{ \oplus \} \). Conversely and, more interestingly, it is possible to prove that, in the presence of binary predicates, in particular in the presence of the equality predicate \( P_=(X, Y) := X.id = Y.id \), we have \( \text{CEL} \cup \{ \oplus \} \not\subseteq \text{CEL} \cup \{ \text{STRICT} \} \).

**Theorem 5.** There exists a set \( \mathcal{P} \) containing a binary CEL predicate and a formula \( \varphi \) in \( \text{CEL} \cup \{ \oplus \}(P) \) that is not equivalent to any formula in \( \text{CEL} \cup \{ \text{STRICT} \}(P) \).

**Proof.** Consider \( \mathcal{P} = \{ P_{=SO} \} \), where \( P_{=}(x, y) := (x.a = y.a) \), and consider the formula:

\[
\varphi = ((A; E) \text{ FILTER } P_{=SO}(A, E)) \oplus
\]

in \( \text{CEL} \cup \{ \oplus \}(P) \). We prove that there is no formula \( \psi \) in \( \text{CEL} \cup \{ \text{STRICT} \}(P) \) equivalent to \( \varphi \). The strategy of our proof will be to define an ad-hoc “pumping lemma” for \( \text{CEL} \cup \{ \text{STRICT} \} \) formulas, to then show that \( \varphi \) does not satisfy such a lemma.

Consider a stream \( S \), a valuation \( \mu \), two positions \( i, j \in \sup(\mu) \) with \( i < j \) and a constant \( k \geq 1 \). Consider the factorization \( \mu_1 \cup \mu_2 \cup \mu_3 \) of \( \mu \) given by \( i, j \) in which, for
every variable \( A \), \( \mu_1(A) \) contains all positions in \( \mu(A) \) lower than \( i \), \( \mu_2(A) \) contains all positions of \( \mu(A) \) between \( i \) and \( j \) (including them) and \( \mu_3(A) \) contains all positions of \( \mu(A) \) higher than \( j \). Likewise, consider the factorization \( S_1 \cdot S_2 \cdot S_3 \) of \( S \) in a similar way: \( S_1 \) contains all events with positions lower than \( i \), \( S_2 \) all events between positions \( i \) and \( j \) (including them) and \( S_3 \) all events with positions higher than \( j \).

Now, we define the result of pumping the fragment \([i, j]\) of \( (S, \mu) \) \( k \) times as a tuple \((S', \mu')\), where \( S' \) and \( \mu' \) are a stream and valuation defined as follows:

- To define \( S' \), we consider two cases. First, when \( \sup(\mu_2) \) does not induce a contiguous interval (that is, there is some \( l \) such that \( i < l < j \) and \( l \notin \sup(\mu_2) \)), we define \( S' \) as \( S_1 \cdot P_0 \cdot S_2 \cdot P_1 \cdot S_2 \cdot \ldots \cdot S_2 \cdot P_k \cdot S_3 \), where \( S_2 \) is repeated \( k \) times and each \( P_i \) is an arbitrary finite stream. Second, when \( \sup(\mu_2) \) does induce a contiguous interval, we define \( S' \) without the \( P_i \), i.e., \( S' = S_1 \cdot S_2 \cdot S_2 \cdot \ldots \cdot S_2 \cdot S_3 \).
- \( \mu' \) is defined as \( \mu_1 \cup \mu_2' \cup \mu_3' \), where each \( \mu_2' \) is the same valuation \( \mu_2 \) but with its values moved to fit the \( i \)-th occurrence of \( S_2 \) in \( S' \). For example, for every variable \( A \), \( \mu_2^3(A) \) results after adding \( |S_2| \) to all values in \( \mu_2(A) \) if it induces a contiguous interval, and adding \( |P_0| + |S_2| + |P_1| \) else. Likewise, \( \mu_3' \) is the same as \( \mu_3 \) but moved to fit \( S_3 \).

Notice that if \( \sup(\mu) \) induces a contiguous interval, then also does \( \sup(\mu') \). Moreover, notice that no new events were added to the valuation, i.e. \( S[\mu(A)] = S'[\mu'(A)] \) for every variable \( A \).

A formula \( \rho \) in CEL is said to be \textit{pumpable} if there exists a constant \( N \in \mathbb{N} \) such that for every stream \( S \), positions \( p_1, p_2 \) and valuation \( \mu \in \|\rho\|((S, p_1, p_2) \) with \( |\sup(\mu)| > N \) there exist two positions \( i, j \in \sup(\mu) \) with \( i < j \) such that for every \( k \geq 1 \) it holds that \( \mu'^k \in \|\rho\|((S', p_1, p_2') \), where \( (S', \mu') \) is the results of pumping the fragment \([i, j]\) of \( (S, \mu) \) \( k \) times and \( p_2' \) is the position at which \( S[p_2] \) ended. In the following lemma we show the utility of this property.

\textbf{Lemma 5.} Every formula \( \varphi \) in CEL\( \cup \{\text{STRICT}\}(\mathcal{P}) \) is pumpable.
PROOF. Consider a formula \( \varphi \) in \( \text{CEL} \cup \{ \text{STRIGT} \} (P) \). We prove the lemma by induction over the structure of \( \varphi \). First, consider the base case \( R \). By defining \( N = 1 \) we know that for every stream \( S \) there is no valuation \( \mu \in \llbracket \varphi \rrbracket (S) \) with \( | \sup(\mu) | > N \), so the lemma holds.

Now, for the inductive step consider first the case \( \varphi = \psi_1 \) \( \text{FILTER} \) \( P(X_1, \ldots, X_n) \). By induction hypothesis, we know that the lemma holds for \( \psi_1 \), thus let \( N_1 \) be its corresponding constant. Let \( N \) be equal to \( N_1 \). Consider any stream \( S \), positions \( p_1, p_2 \) and valuation \( \mu \in \llbracket \varphi \rrbracket (S, p_1, p_2) \) with \( | \sup(\mu) | > N \). By definition \( \mu \in \llbracket \psi_1 \rrbracket (S, p_1, p_2) \) and \( \llbracket \psi_1 \rrbracket (S[\mu(X_1)], \ldots, S[\mu(X_n)]) \in P \). By induction hypothesis, \( \psi_1 \) is pumpable, thus there exist positions \( i, j \in \sup(\mu) \) with \( i < j \) such that the fragment \([i, j]\) can be pumped. Moreover, consider that the result of pumping the fragment \([i, j]\) \( k \) times is \( (S', \mu') \), for an arbitrary \( k \). Then, it holds that \( \mu' \in \llbracket \varphi \rrbracket (S'p_1, p_2') \).

Also, because in the pumping it holds that \( S[\mu(A)] = S'[\mu'(A)] \) for every \( A \), then \( (S'[\mu'(X_1)], \ldots, S'[\mu'(X_n)]) \in P \). Therefore, \( \mu' \in \llbracket \varphi \rrbracket (S'p_1, p_2') \), thus \( \varphi \) is pumpable.

Consider now the case \( \varphi = \psi_1 \) \( \text{OR} \) \( \psi_2 \). By induction hypothesis, we know that the property holds for \( \psi_1 \) and \( \psi_2 \), thus let \( N_1 \) and \( N_2 \) be the corresponding constants, respectively. Then, we define the constant \( N \) as the maximum between \( N_1 \) and \( N_2 \). Consider any stream \( S \), positions \( p_1, p_2 \) and valuation \( \mu \in \llbracket \varphi \rrbracket (S, p_1, p_2) \) with \( | \sup(\mu) | > N \). By definition or \( \text{OR} \), either \( \mu \in \llbracket \psi_1 \rrbracket (S, p_1, p_2) \) or \( \mu \in \llbracket \psi_2 \rrbracket (S, p_1, p_2) \), so w.l.o.g. consider the former case. By induction hypothesis, \( \psi_1 \) is pumpable, thus there exist positions \( i, j \in \sup(\mu) \) with \( i < j \) such that the fragment \([i, j]\) can be pumped and the result \((S', \mu')\) satisfies \( \mu' \in \llbracket \varphi \rrbracket (S', p_1, p_2') \). This means that \( \mu' \in \llbracket \varphi \rrbracket (S'p_1, p_2') \), therefore, \( \varphi \) is pumpable.

Now, consider the case \( \varphi = \psi_1 ; \psi_2 \). By induction hypothesis, we know that the property holds for \( \psi_1 \) and \( \psi_2 \), thus let \( N_1 \) and \( N_2 \) be the corresponding constants, respectively. Then, we define the constant \( N = N_1 + N_2 \). Consider any stream \( S \), positions \( p_1, p_2 \) and valuation \( \mu \in \llbracket \varphi \rrbracket (S, p_1, p_2) \) with \( | \sup(\mu) | > N \). This means that there exist \( p' \in \mathbb{N} \) and valuations \( \mu_1, \mu_2 \) such that \( \mu = \mu_1 \cup \mu_2, \mu_1 \in \llbracket \psi_1 \rrbracket (S, p_1, p') \) and \( \mu_2 \in \llbracket \psi_2 \rrbracket (S, p' + 1, p_2) \). Moreover, either \( | \sup(\mu_1) | > N_1 \) or \( | \sup(\mu_2) | > N_2 \), so w.l.o.g. assume the former case. By induction hypothesis, \( \psi_1 \) is pumpable, thus
there exist positions $i, j \in \sup(\mu_1)$ with $i < j$ such that the fragment $[i, j]$ can be pumped and the result $(S', \mu'_1)$ satisfies $\mu'_1 \in \| \psi_1 \|(S', p_1, p' + r)$, assuming that the pumping added $r$ new events. Define the valuation $\mu' = \mu'_1 \cup \mu'_2$, where $\mu'_2$ is the same as $\mu_2$ but adding $r$ to each value (so that $(S[\sup(\mu_2)] = S'[\sup(\mu'_2)])$. Then $\mu'_1 \in \| \psi_1 \|(S', p_1, p' + r)$ and $\mu'_2 \in \| \psi_2 \|(S', p' + r + 1, p_2 + r)$, thus $\mu' \in \| \varphi \|(S', p_1, p_2 + r)$, therefore, $\varphi$ is pumpable.

Consider then the case $\varphi = \psi_1 +$. By induction hypothesis, we know that the lemma holds for $\psi_1$, thus let $N_1$ be its corresponding constant. Let the constant $N$ be equal to $N_1$. Consider any stream $S$, positions $p_1, p_2$ and valuation $\mu \in \| \varphi \|(S, p_1, p_2)$ with $|\sup(\mu)| > N$. Then, consider $i = \min(\sup(\mu))$ and $j = \max(\sup(\mu))$, consider any $k \geq 1$ and let $(S', \mu')$ be the result of pumping the fragment $[i, j]$ of $(S, \mu)$ $k$ times. We prove now that $\mu' \in \| \varphi \|(S', p_1, p'_2)$ by induction over $k$. If $k = 1$ then, as defined in the definition of pumping, $S'$ has the form $S_1 \cdot P_0 \cdot S_2 \cdot P_1 \cdot S_3$, and $\mu'$ is the same as $\mu$ but adding $r$ to each position, where $r = |P_0|$. Clearly it holds that $\mu' \in \| \varphi \|(S', p_1, p'_2)$, since the modifications did not affect the part of $S$ in the valuation. Now, consider that $k > 1$. Then, $S'$ has the form $S_1 \cdot P_0 \cdot S_2 \cdot P_1 \cdot S_2 \cdot . . . S_2 \cdot P_k \cdot S_3$. Similarly, $\mu'$ is defined as $\mu_1 \cup \mu_2^1 \cup \mu_2^2 \cup \ldots \cup \mu_2^k \cup \mu_3$, where $\mu_1(A) = \mu_3(A) = 0$ for any variable $A$, and each $\mu_2^i$ is the same valuation $\mu$ but with their positions moved to fit the $i$-th occurrence of $S_2$ in $S'$. Consider that $r = |S_1 \cdot P_0 \cdot S_2|$. By induction hypothesis, we can say that $\mu'_2 \in \| \varphi \|(S', r + 1, p'_2)$ where $\mu'_2 = \mu_2^1 \cup \ldots \cup \mu_2^k$ (notice that we consider it from position $r + 1$ because there is no lower position in $\sup(\mu'_2)$). Also, it is easy to see that this implies $\mu'_2 \in \| \varphi + \|(S', r + 1, p'_2)$, which is something we will need next. Moreover, it holds that $\mu_1^2 \in \| \varphi \|(S', p_1, r)$, because it represents the same valuation as the original one $\mu$. Then, because $\mu' = \mu_1^2 \cup \mu'_2$, it follows that $\mu' \in \| \varphi ; \varphi + \|(S', p_1, p'_2)$ which also implies that $\mu' \in \| \varphi + \|(S', p_1, p'_2)$. Since $\varphi + = (\psi_1 +) \equiv \psi_1 + = \varphi$, it holds that $\mu' \in \| \varphi \|(S', p_1, p'_2)$.

Now, consider the case $\varphi = \text{STR\textsc{ict}}(\psi_1)$. By induction hypothesis, we know that the lemma holds for $\psi_1$, thus let $N_1$ be its corresponding constant. Let the constant $N$ for $\varphi$ be equal to $N_1$. Consider any stream $S$, positions $p_1, p_2$ and valuation $\mu \in \| \varphi \|(S, p_1, p_2)$ with $|\sup(\mu)| > N$. Then, by definition $\mu \in \| \psi_1 \|(S, p_1, p_2)$, and
by induction hypothesis there exist positions $i, j \in \sup(\mu)$ such that the fragment $[i, j]$ can be pumped and the result $(S', \mu')$ satisfies $\mu' \in \llbracket \psi_1 \rrbracket(S, p_1, p_2)$. Notice that $\sup(\mu)$ induces a contiguous interval because of the definition of STRICT, therefore $\sup(\mu')$ also induces a contiguous interval, thus $\mu' \in \llbracket \varphi \rrbracket(S, p_1, p_2)$.

Finally, consider the case $\varphi = \psi_1 \text{ IN } A$. By induction hypothesis, we know that the lemma holds for $\psi_1$, thus let $N_1$ be its corresponding constant. Let the constant $N$ be equal to $N_1$. Consider any stream $S$, positions $p_1, p_2$ and valuation $\mu \in \llbracket \varphi \rrbracket(S, p_1, p_2)$ with $|\sup(\mu)| > N$. Then, by definition there exists $\eta$ such that $\eta \in \llbracket \psi_1 \rrbracket(S, p_1, p_2)$ and $\mu = \eta[A \rightarrow \sup(\eta)]$. By induction hypothesis there exist positions $i, j \in \sup(\eta)$ such that the fragment $[i, j]$ can be pumped and the result $(S', \eta')$ satisfies $\eta' \in \llbracket \psi_1 \rrbracket(S, p_1, p_2)$. Note that the results $(S', \mu')$ and $(S', \eta')$ of pumping $[i, j]$ in $(S, \mu)$ and $(S, \eta)$, respectively, are the same, with the only difference that $\mu'$ satisfies $\mu'(A) = \sup(\eta')$. Then it follows that $\mu' \in \llbracket \varphi \rrbracket(S, p_1, p_2)$. \hfill \Box

The last ingredient is to show that there is no formula $\psi$ in $\text{CEL} \cup \{\text{STRICT}\}(P)$ that is equivalent to $\varphi = ((A; E) \text{ FILTER } P^{SO}(A, E)) \oplus$ by proving that such formula is not pumpable. By contradiction, assume that $\psi$ exists, and let $N$ be its constant. Consider then the stream:

$$S = \begin{array}{cccccccc}
A & L & E & A & L & E & \cdots & A & L & E & \cdots \\
1 & 1 & 1 & 2 & 2 & 2 & & N & N & N & \\
\end{array}$$

Where the first and second lines are the type and $a$ attribute of each event, respectively, and consider the valuation $\mu$ with $\mu(A) = \{1, 4, 7, \ldots, 3N - 2\}$ and $\mu(E) = \{3, 6, 9, \ldots, 3N\}$. Now, let $i, j$ be any two positions of $\sup(\mu)$, which define the partitions $\mu_1 \cup \mu_2 \cup \mu_3$, and name $t_1 = S[i]$ and $t_2 = S[j]$. We will use $k = 2$, i.e., repeat section $S[i, j]$ two times, and use the 1-tuple stream $U(0)$ as the arbitrary streams $P_0$ and $P_1$ to get the resulting stream $S'$ and the corresponding valuation $\mu'$. We will analyse the following possible cases: type$(t_1) = \text{type}(t_2)$; type$(t_1) = A$ and type$(t_2) = E$; type$(t_1) = E$ and type$(t_2) = A$. In the first case the resulting $\mu'$ is a valuation with two consecutive tuples of the same type, which contradicts the original formula $\varphi$. In the second case $\sup(\mu_2)$ is not a contiguous interval so the valuation $\mu'$ would fail to
ensure that the $A$ tuple following $t_2$ is placed right after it (because of the tuple $U(0)$ inbetween), thus contradicting the $\oplus$ property of $\varphi$. In the third case it is clear that the last $A$ in the first repetition of $[i, j]$ and the first $E$ in the second repetition (i.e., $S[j]$ and $S[j + 2]$) do not satisfy the FILTER condition because $S[j].a > S[j + 2].a$. Finally, the formula $\psi$ cannot exist.

From this result, it follows that the containment $\text{CEL} \cup \{\text{STRICT}\}(P) \subset \text{CEL} \cup \{:, \oplus\}(P)$ is strict in that case.

The final question we address in this section is what happens when only unary predicates are considered. This question fits naturally in the presented work, since the restriction to unary predicates has already been presented and studied in the previous sections. Interestingly, for this particular case, $\text{STRICT}$ can be shown to be just as expressive as $:$ and $\oplus$ together.

**Proposition 6.** For a set of unary CEL predicates $U$, $\text{CEL} \cup \{\text{STRICT}\}(U) \equiv \text{CEL} \cup \{:, \oplus\}(U)$; that is, it holds that $\text{CEL} \cup \{\text{STRICT}\}(U) \subseteq \text{CEL} \cup \{:, \oplus\}(U)$ and $\text{CEL} \cup \{:, \oplus\}(U) \subseteq \text{CEL} \cup \{\text{STRICT}\}(U)$.

**Proof.** Consider a formula $\varphi$ in $\text{CEL} \cup \{:, \oplus\}(U)$. We first prove that there is a $\psi$ in $\text{CEL} \cup \{:, \text{STRICT}\}(U)$ which is equivalent to $\varphi$, and then the proof follows directly from Proposition 4.

Consider any formula $\varphi'$ in $\text{CEL} \cup \{:, \text{STRICT}\}(U)$. We prove by induction that there exists a formula $\psi'$ in $\text{CEL} \cup \{:, \text{STRICT}\}(U)$ which is equivalent to $\varphi' \oplus$. For simplicity, we assume that all the filters are applied at the bottom-most level of the formula and that it has no labels. Every formula can be translated to have this property by pushing down the filter of each subformula $\sigma \text{FILTER } P(A)$ with the following procedure:

(i) Push down variable $A$ as shown in the proof of Proposition 4, and
(ii) Remove the filter and replace each subformula of $\sigma$ with the form $R \text{IN } A$ by $R \text{FILTER } P(R)$. 


If \( A \) is a relation name, then just remove the filter and replace each subformula \( A \) by \( A \text{ FILTER } P(A) \).

Now that all filters in our \( \phi' \) are at the bottom-most level and all the IN are dropped, we prove by induction that there exists a formula \( \psi' \) in \( \text{CEL} \cup \{ :, \text{STRICT} \} (U) \) which is equivalent to \( \phi' \oplus \). We consider the possible cases for \( \phi' \):

- For the base case, if \( \phi' = R \), then \( \psi' = \text{STRICT}(R+) \) is equivalent to \( \phi' \oplus \).
  Similarly for the case \( \phi' = R \text{ FILTER } P(R) \).
- If \( \phi' = \text{STRICT}(\phi_1) \), then \( \psi' = \text{STRICT}(\phi_1+) \) is equivalent to \( \phi' \oplus \).
- If \( \phi' = \phi_1 ; \phi_2 \), then \( \psi' = \phi_1 ; (\phi_2 \text{ OR } ((\phi_2 : \phi_1)+ ; \phi_2)) \) is equivalent to \( \phi' \oplus \).
- If \( \phi' = \phi_1+ \), then \( \psi' = \phi_1+ \) is equivalent to \( \phi' \oplus \).

We do not consider the : -case since we know that they can be removed by using \text{STRICT} instead.

The last and more complex operator is the OR, for which we have to consider \( \phi' = \phi_1 \text{ OR } \phi_2 \), with all possible cases for \( \phi_1 \) and \( \phi_2 \). The simplest scenario is where both \( \phi_1 \) and \( \phi_2 \) have either the form \( R, R \text{ FILTER } P(R) \) or \( \text{STRICT}(\psi) \) for some \( \psi \), at which case we can simply write \( \phi' \oplus \) as \( \text{STRICT}(\phi'+) \).

Now we consider the cases where some of them does not have this form (w.l.o.g. assume is \( \phi_2 \)). Consider first the case \( \phi_2 = \rho_1 ; \rho_2 \). Here we use the following equivalence:

\[
(\phi_1 \text{ OR } (\rho_1 ; \rho_2)) \oplus \equiv \phi_1 \oplus (\rho_1 ; \rho_2) \oplus \text{ OR} \tag{1}
\]

\[
(\phi_1 \oplus : (\rho_1 ; \rho_2) \oplus ) \oplus \text{ OR} \tag{2}
\]

\[
\phi_1 \oplus : ((\rho_1 ; \rho_2) \oplus : \phi_1 \oplus) \oplus \text{ OR} \tag{3}
\]

\[
((\rho_1 ; \rho_2) \oplus : \phi_1 \oplus) \oplus \text{ OR} \tag{4}
\]

\[
(\rho_1 ; \rho_2) \oplus : (\phi_1 \oplus : (\rho_1 ; \rho_2)) \oplus \tag{5}
\]

Here, part (1) has no problem since the \( \oplus \)-operator is applied over subformulas of the original one, thus by induction hypothesis they can be written without \( \oplus \). Moreover, with some basic transformations in part (2) (namely, replacing \( (\rho_1 ; \rho_2) \oplus \) with \( \rho_1 ; (\rho_2 \text{ OR } ((\rho_2 : \rho_1)+ ; \rho_2)) \)) one can show that it is equivalent to the formula \( (\sigma_1 ; \sigma_2) \oplus \),
where \( \sigma_1 = \varphi_1 \oplus : \rho_1 \) and \( \sigma_2 = \rho_2 \mathcal{O} \mathcal{R} (\rho_2 : \rho_1)\mathcal{O} \mathcal{R} (\rho_2 : \rho_1)\mathcal{O} \mathcal{R} (\rho_2 : \rho_1)\mathcal{O} \mathcal{R} (\rho_2 : \rho_1)\). Then, we can replace \((\sigma_1 ; \sigma_2)\) with \(\sigma_1 ; (\sigma_2 \mathcal{O} \mathcal{R} ((\sigma_2 : \sigma_1)\mathcal{O} \mathcal{R} \sigma_2))\), and the resulting formula will contain only one \(\oplus\) in the form \(\varphi_1 \oplus\), which by induction hypothesis can also be removed. Similarly, parts (3), (4) and (5) can be rewritten this way, therefore for the case of \(\varphi_2 = \rho_1 ; \rho_2\) the induction statement remains true.

Now consider the case \(\varphi_2 = \rho_+\). Notice that the following equivalence regarding \(+\) holds: \(\rho_+ \equiv \rho \mathcal{O} \mathcal{R} \rho ; \rho_+\). Thus, \(\varphi'\) can then be written as \((\varphi_1 \mathcal{O} \mathcal{R} \rho) \mathcal{O} \mathcal{R} (\rho ; \rho+)\). Then, if we redefine \(\varphi_1 := (\varphi_1 \mathcal{O} \mathcal{R} \rho)\) and \(\varphi_2 = (\rho ; \rho+)\), clearly \(\varphi'\) would have the form \(\varphi_1 \mathcal{O} \mathcal{R} (\rho_1 ; \rho_2)\). Therefore, we can apply the previous case and the resulting formula will still satisfy the induction statement, thus it remains true in the case \(\varphi_2 = \rho_+\).

Then, we can replace every subformula \(\varphi' \oplus\) of \(\varphi\) with its equivalent formula \(\psi'\) in a bottom-up fashion to ensure that each \(\varphi'\) is in \(\text{CEL} \cup \{ : , \mathcal{O} \mathcal{R} \text{STRICT} \}(\mathcal{U})\). Finally, the remaining formula \(\psi\) does not contain \(\oplus\) and thus is in \(\text{CEL} \cup \{ : , \mathcal{O} \mathcal{R} \text{STRICT} \}(\mathcal{U})\). The converse case follows directly from Proposition 5.

This last proposition concludes our discussion on the operators for contiguity, and allows us to argue that including the operators : and \(\oplus\) is better than including the unary operator \(\mathcal{O} \mathcal{R} \text{STRICT}\), at least in terms of expressiveness. However, the decision depends mostly on the nature of the predicates that are going to be supported.

3.6. Algorithms for Evaluating CEA

In this section we show how to efficiently evaluate CEA. More specifically, we formally present the problem of CEA Evaluation, and then describe an algorithm, with an ad-hoc data structure, that solves this problem with linear update time and output-linear delay enumeration.

**CEA Evaluation Problem.** Given a unary CEL formula \(\varphi\) and a stream \(S = t_0 t_1 \ldots\) we want to evaluate \(\varphi\) over \(S\) by reading the events of \(S\) in order and, after each new read event \(t_n\), computing \(\llbracket \varphi \rrbracket_n(S)\). Because \(\varphi\) can be compiled into a CEA
$\mathscr{A}_\phi$ (cf. Section 3.3), we can reduce this problem to computing $[\mathcal{A}_\phi]_n(S)$. This hence yields the following enumeration problem.

**Problem:** EVALUATIONCEA  
**Input:** A CEA $\mathcal{A}$ and a stream $S = t_0t_1\ldots$  
**Output:** Enumerate the set $[\mathcal{A}]_n(S)$ after reading $t_n$, for every $n \geq 0$

Having formalized the EVALUATIONCEA problem, we proceed to show how to solve it with the above guarantees. Specifically, in the rest of this section we show the following result.

**Theorem 6.** For every CEA $\mathcal{A}$ in normal form, there is a CER evaluation algorithm with $|A|$-update time and output-linear delay.

Here, CEA $\mathcal{A}$ is in normal form if (1) it is single-variable: there exists $A \in L$ such that in every transition $(p, P, L, q)$ of $\mathcal{A}$, either $L = \{A\}$, or $L = \emptyset$, and (2) it is I/O deterministic. We already know from Proposition 1 that every CEA can be I/O determinized. The following proposition shows that, as far as the complex event semantics of CEA is concerned, every CEA can also be converted into a single-variable one. Recall that two CEA $\mathcal{A}_1$ and $\mathcal{A}_2$ are complex event equivalent (denoted $\mathcal{A}_1 \equiv_c \mathcal{A}_2$) if for every stream $S$ and every index $n$ we have $[\mathcal{A}_1]_n(S) = [\mathcal{A}_2]_n(S)$.

**Proposition 7.** For every CEA $\mathcal{A}$ there exists a single-variable CEA $\mathcal{A}'$ such that $\mathcal{A} \equiv_c \mathcal{A}'$. Moreover, the size of $\mathcal{A}'$ is linear in the size of $\mathcal{A}$.

**Proof.** Fix an arbitrary variable $A \in L$. Let $\mathcal{A}'$ be obtained from $\mathcal{A}$ by replacing each transition $(p, P, L, q)$ by the transition $(p, P, \{A\}, q)$ if $L \neq \emptyset$, and by the transition $(p, P, \emptyset, q)$ otherwise. Note that a run $\rho'$ of $\mathcal{A}'$ will mark an event by $A$ if, and only if, a corresponding run $\rho$ of $\mathcal{A}$ exists that marks this event by a non-empty set of variables. Therefore, $\sup(\mu_{\rho'}) = \sup(\mu_{\rho})$. Hence $[\mathcal{A}]_n(S) = [\mathcal{A}']_n(S)$ for every stream $S$ and every position $n$. \hfill $\square$

Since we can always convert an arbitrary CEA into normal form by first converting it in an equivalent single-variable CEA and then I/O-determinizing it\(^1\), the algorithm

\(^1\)Note that the I/O-determinizing a single-variable CEA using the method of Proposition 1 will result again in a single-variable CEA.
of Theorem 6 immediately yields a CER evaluation algorithm for arbitrary CEA (not necessarily in normal form). Unfortunately, the determinization procedure has an exponential blow-up in the size of the CEA in the worst case, leading to the following result.

**Corollary 5.** For every CEA $A$, there is a CER evaluation algorithm with $2^{|A|}$-update time and output-linear delay.

We can further extend the CER evaluation algorithm for CEA to any selection strategy by using the results of Theorem 3.

The rest of this section is devoted to proving Theorem 6. We start by defining the data structure of our evaluation algorithm. Then, we explain how to store sets of complex events in this data structure and how to enumerate them with output-linear delay. After this, we present how to update the data structure when new events arrive, and finish with the correctness proof. We start, however, with some notation and intuition.

**Notation.** By definition, any transition of a CEA $A$ in normal form will be of the form $(p, P, L, q)$ with $L$ either the singleton $\{A\}$ for some variable $A$, or the empty set. The former transition kind marks input events whereas the latter does not. In what follows, we find it convenient to visually emphasize that a transition is marking or not by writing $(p, P, \bullet, q)$ instead of $(p, P, L, q)$ when $L \neq \emptyset$ and $(p, P, \circ, q)$ when $L = \emptyset$. Hence, $(p, P, \bullet, q)$ indicates a marking transition from $p$ to $q$, and $(p, P, \circ, q)$ a non-marking one. Throughout the rest of the section, we assume that the CEA $A$ to be evaluated is in normal form.

**Intuition behind the data structure.** Our data structure will be a directed acyclic graph (DAG) that compactly represents sets of complex events. Specifically, this DAG will be endowed with a special terminator node $\bot$, and every path to $\bot$ will encode a complex event. To illustrate how this works, consider the set of complex events $C = \{\{5, 2, 1\}, \{5, 3, 1\}, \{5, 3\}, \{6, 2, 1\}, \{6, 3, 1\}, \{6, 3\}, \{6, 4\}\}$. This set is compactly represented by the dag $DAG_{\mathcal{G}}$ that is depicted at the right of Fig 3.7: every complex event $C \in C$ corresponds to a path from node 5 or 6 in $DAG_{\mathcal{G}}$ to $\bot$. Note that all
nodes in \( G_D \), except \( \perp \) are labeled by positions (i.e., elements of \( \mathbb{N} \)). Multiple nodes may be labeled by the same position (not shown in Fig. 3.7).

We will construct the DAG in such a way that (1) every node has a path to \( \perp \), (2) no path to \( \perp \) ever repeats a position and (3) distinct paths to \( \perp \) represent distinct complex events. As such, we may enumerate \( \mathcal{C} \) from \( G_D \) without repetitions simply by enumerating all paths from nodes 5 and 6 to \( \perp \) in \( G_D \). This can be done with output-linear delay using depth-first search, as we will see.

For the purpose of maintaining the data structure in the desired time \( O(|A| \cdot W_U(|t|)) \) when processing a new event \( t \), we will actually need to be smart about how we represent the set of outgoing edges of a node \( n \). Note in particular that this update time must be independent of the size of the data structure itself. When adding a new node to the data structure, the time spent connecting this new node to its children must hence be independent of the number of children to connect to. Therefore, rather than simply connect \( n \) to all of its children (of which there may be unboundedly many), we will arrange the children in a linked list of nodes, and connect \( n \) to the first and last node in this linked list, respectively. Assuming that the linked list itself has already been built, connecting \( n \) to the linked list only involves setting the first and last pointers, which is a constant time operation. It then suffices to define the update phase algorithm in such a way that it builds the linked lists incrementally.

We may see the linked lists, as well as the pointers to the first and last nodes in these lists, as yet another DAG—this time with three kinds of edges: next edges that connect a node to its successor in the linked list; first edges that connect a node to the first node in the linked list of its children; and last edges that connect a node to the last node in that linked list. To illustrate, the encoding in this way of the DAG \( G_D \) of Fig. 3.7 (right) is shown as the DAG \( \mathcal{D} \) in Fig. 3.7 (left). There, next edges are depicted by dashed lines, while first and last edges are depicted as solid lines. It is this DAG \( \mathcal{D} \), which we call a \textit{Compact Complex Event Set}, that forms our data structure.
The data structure. Formally, we define a Compact Complex Event Set (CCES for short) as a tuple:

\[ D = (N, \perp, \text{event}, \text{first}, \text{last}, \text{next}) \] (*)

where \(N\) is a finite set of nodes, \(\perp \in N\) is a special node, \(\text{event}: N \setminus \{\perp\} \to N\) is a function that maps nodes to events (i.e., positions), \(\text{first}\) and \(\text{last}\) are functions from \(N \setminus \{\perp\}\) to \(N\), and \(\text{next}\) is a partial function from \(N\) to \(N\). For brevity, we write \(\text{next}(n) = \emptyset\) when \(\text{next}\) is not defined over \(n\).

To be valid, a CCES must satisfy the following five restrictions.

Let \(\text{Next}_D\) be the subgraph of \(D\) formed by only the next edges, i.e., \(\text{Next}_D = (N, \{(n, \text{next}(n)) \mid n \in N \land \text{next}(n) \neq \emptyset\})\). The first two restrictions that \(D\) must satisfy are:

(i) The graph \(\text{Next}_D\) is acyclic.

(ii) For every \(n \in N\), there is a path from \(\text{first}(n)\) to \(\text{last}(n)\) in \(\text{Next}_D\).

Given that \(\text{next}\) is a partial function, (1) implies that \(\text{Next}_D\) consists of a set of paths. Furthermore, (2) implies that, for each \(n\), the pair \((\text{first}(n), \text{last}(n))\) represents a list of nodes, starting in \(\text{first}(n)\) and ending in \(\text{last}(n)\). In the middle of Fig. 3.7, we display the graph \(\text{Next}_D\) for our example. The reader can check that the aforementioned properties are satisfied.
We say that \( n \in \mathbb{N} \) can reach \( n' \in \mathbb{N} \) if, and only if, there is a (possibly empty) directed path from \( n \) to \( n' \) in \( \text{Next}_D \). If this is the case, we define \( \text{list}(n, n') = n_0, \ldots, n_k \) such that \( n_0 = n, n_{i+1} = \text{next}(n_i) \), and \( n_k = n' \). By Property (2) we know that, for every \( n \in \mathbb{N} \), \( \text{first}(n) \) can reach \( \text{last}(n) \) and, moreover, \( \text{list} ( \text{first}(n), \text{last}(n) ) \) is the list of nodes between them. By some abuse of notation, we will write \( n'' \in \text{list}(n, n') \) to denote that \( n'' \) appears in \( \text{list}(n, n') \).

For the third restriction, define the directed graph:

\[
G_D = (\mathbb{N}, \{(n, n') \mid n' \in \text{list} ( \text{first}(n), \text{last}(n) ) \}).
\]

Then, every CCES \( \mathcal{D} \) must also satisfy the following property:

(3) The graph \( G_D \) is acyclic.

In fact, as already mentioned, the purpose of \( \mathcal{D} \) is to represent the acyclic graph \( G_D \). Note that \( \bot \) is the only node without out-edges in \( \mathcal{D} \) (i.e., \( \text{first} \) and \( \text{last} \) is not defined for \( \bot \)) and, hence, by the acyclicity of \( G_D \), every path of \( G_D \) ends in \( \bot \). For an illustration, in Fig. 3.7 (right) we provide the acyclic graph \( G_D \) for our example.

Until now we have not used the event function of CCES \( \mathcal{D} \), which is used to represent complex events. Let \( \pi = n_0, \ldots, n_k \) be a path in \( G_D \) ending in \( \bot \) (i.e., \( n_k = \bot \)). Let the complex event associated to \( \pi \) be \( \text{CE}(\pi) = \{\text{event}(n_0), \ldots, \text{event}(n_{k-1})\} \), and for every \( n \in \mathbb{N} \) let \( \text{CE}(n) = \{\text{CE}(\pi) \mid \pi \text{ is a path in } G_D \text{ starting in } n \text{ and ending in } \bot \} \). Furthermore, for every \( n, n' \in \mathbb{N} \) such that \( n \) can reach \( n' \) we define \( \text{CE}(n, n') = \bigcup_{n'' \in \text{list}(n, n')} \text{CE}(n'') \). In our running example, we can check that \( \text{CE}(5) \) is equal to \( \{\{1, 2, 5\}, \{1, 3, 5\}, \{3, 5\}\} \), which are all paths from 5 to \( \bot \) (see Fig. 3.7, right). One can also check that, for example, \( \text{CE}(2, 4) = \{\{1, 2\}, \{1, 3\}, \{3\}, \{4\}\} \).

While the graph \( G_D \) represented by \( \mathcal{D} \) is of polynomial size with respect to \( \mathcal{D} \), it can encode an exponential number of complex events. To enumerate them efficiently, we need to impose the last two restrictions on \( \mathcal{D} \):

(4) For every \( n \in \mathbb{N} \setminus \{\bot\} \) and \( n' \in \text{list} ( \text{first}(n), \text{last}(n) ) \), if \( n' \neq \bot \), then \( \text{event}(n) > \text{event}(n') \).
Algorithm 1 Given a Compact Complex Event Set $\mathcal{D}$, two nodes $n_1$ and $n_2$ such that $n_1$ can reach $n_2$, and a complex event $C$, it enumerates $\bigcup_{C' \in \text{CE}(n_1, n_2)} \{C \cup C'\}$.

1: procedure ENUMERATE($\mathcal{D}$, $n_1$, $n_2$, $C$)
2: \quad if $n_1 = \perp$ then
3: \quad \quad Output($C$)
4: \quad else
5: \quad \quad ENUMERATE($\mathcal{D}$, first($n_1$), last($n_1$), $C \cup \{\text{event($n_1$)}\}$)
6: \quad \quad if $n_1 \neq n_2$ then
7: \quad \quad \quad ENUMERATE($\mathcal{D}$, next($n_1$), $n_2$, $C'$)

(5) For every $n, n' \in \mathbb{N}$ such that $n$ can reach $n'$, and for every pair of different paths $\pi$ and $\pi'$ starting in some node in list($n, n'$) and ending in $\perp$, it holds that $\text{CE}(\pi) \neq \text{CE}(\pi')$.

Property (4) forces that the (positions of the) events of a complex event represented by a path $\pi$ in $G_D$ are in decreasing order and, therefore, positions cannot be repeated in a path. Property (5) enforces that there are no “repetitions” in the set $\text{CE}(n, n')$, namely, two paths that define the same complex event. This fact will allow us to enumerate the complex events in $\text{CE}(n, n')$, one by one, without repetitions, and with output-linear delay. The reader can verify that restrictions (4) and (5) are satisfied by our CCES example of Fig. 3.7.

**Enumeration phase.** Let $\mathcal{D}$ be a CCES that satisfies properties (1) to (5) and let $n_1$ and $n_2$ be nodes in $\mathcal{D}$ such that $n_1$ can reach $n_2$. This pair $(n_1, n_2)$ encodes the set of complex events $\text{CE}(n_1, n_2)$.

To enumerate $\text{CE}(n_1, n_2)$ with output-linear delay, we provide Algorithm 1. The procedure ENUMERATE receives as input $\mathcal{D}$, $n_1$ and $n_2$, and a complex event $C$. As output, the procedure prints all complex events $\bigcup_{C' \in \text{CE}(n_1, n_2)} \{C \cup C'\}$. In fact, ENUMERATE is a recursive procedure and $C$ is used to store the final output, that is passed to the next call to extend it with complex events in $\text{CE}(n_1, n_2)$. As a special case, the initial call to ENUMERATE will set $C = \emptyset$ and the output will be $\text{CE}(n_1, n_2)$.

Recall the graph $G_D$ represented by $\mathcal{D}$. Algorithm 1 enumerates $\text{CE}(n_1, n_2)$ by traversing $G_D$ recursively. For the base case, when $n_1 = \perp$, the algorithm reaches the sink node of $G_D$ and it prints $C$. For the recursive case, it iterates with $n$ over
the nodes of list \( \text{list}(n_1, n_2) \) and enumerates \( \mathcal{CE}(n) = \{ \text{event}(n) \cup C' \mid C' \in \mathcal{CE}(\text{first}(n), \text{last}(n)) \} \). In other words, \textsc{Enumerate} does a depth-first search of all paths of \( G_D \) that start from some node in \( \text{list}(n_1, n_2) \). The events of the path are stored in \( C \), and each time that the sink node of \( G_D \) (i.e., \( \perp \)) is reached, \( C \) is output. The correctness of this procedure follows directly from its definition.

To verify that the algorithm prints all complex events in \( \mathcal{CE}(n_1, n_2) \) with output-linear delay, one has to notice two facts. First, recall that \( D \) satisfies Property (5), which means that \( \mathcal{CE}(\pi) \neq \mathcal{CE}(\pi') \) for every pair of different paths \( \pi \) and \( \pi' \) starting in some node in \( \text{list}(n_1, n_2) \) and ending in \( \perp \). Therefore, the enumeration of complex events gives no repetitions. Second, after \( C \) is printed by \textsc{Enumerate} (line 3), the recursion performs a “backtracking” until the next node that satisfies \( n_1 \neq n_2 \) (line 6). Then the recursion calls \textsc{Enumerate} going into \( G_D \) until \( \perp \) is reached again. The number of backtracking steps is at most the size of the last complex event that was output, and the number of steps to \textsc{Enumerate} the next complex event depends on the size of the new output. In total, the delay between the last output \( C \) and the next output \( C' \) depends just on \( |C| \) and \( |C'| \). Strictly speaking, this is not output-linear delay, given that the delay should be bounded just on \( C' \). One can see that by delaying the print of \( \# \) (i.e., the separator between outputs, see Section 2.3) that ends \( C \) until the backtracking is done, then the delay will only depend on the next output \( C' \). Thus, we conclude that Algorithm 1 enumerates \( \mathcal{CE}(n_1, n_2) \) with output-linear delay as expected.

**Operations over the data structure.** For the update phase, we need some operations to manage our CCES \( D \). Each operation should take constant time so that the update phase takes time proportional to \( \mathcal{CE}_A \) regardless of the size of \( D \). The operations needed are three:

\[
(D, \perp) \ := \ \text{init}() \\
(D', n) \ := \ \text{extend}(D, i, n_1, n_2) \quad \text{s.t.} \ n_1 \ \text{can reach} \ n_2 \\
D' \ := \ \text{append}(D, n_1, n_2) \quad \text{s.t.} \ \text{next}(n_1) = \emptyset
\]

The first operation is to initialize a new CCES \( D \) with its corresponding special node \( \perp \). The other two operations receive a CCES \( D = (N, \perp, \text{event}, \text{first}, \text{last}, \text{next}) \) and output an extension of \( D \), denoted as \( D' = (N', \perp, \text{event'}, \text{first'}, \text{last'}, \text{next'}) \).
The extend operation receives as input a CCES $\mathcal{D}$, an event $i$ (i.e., a position), and two nodes $n_1$ and $n_2$ of $\mathcal{D}$ such that $n_1$ can reach $n_2$ (i.e., they represent a list). As output, it gives $\mathcal{D}'$, which is $\mathcal{D}$ extended with a fresh node $n$ such that $\text{event}'(n) = i$, $\text{first}'(n) = n_1$, $\text{last}'(n) = n_2$, and $\text{next}'(n)$ is not defined (i.e., $\text{next}'(n) = \emptyset$). In particular, it holds that $N' = N \cup \{n\}$ and $f'(n') = f(n')$ for every $f \in \{\text{event, first, last}\}$ and $n' \in N$.

Intuitively, append will be used to concatenate one list with another. Recall that a pair $(n_1, n_2)$ such that $n_1$ can reach $n_2$ represents the list of nodes $\text{list}(n_1, n_2)$. Then, if we have two pairs $(n_1, n_2)$ and $(n'_1, n'_2)$ such that $\text{next}(n_2) = \emptyset$, after applying $\text{append}(\mathcal{D}, n_2, n'_1)$, $n'_2$ will be reachable from $n_1$ in $\mathcal{D}'$, and $(n_1, n'_2)$ will be the concatenation of $\text{list}(n_1, n_2)$ and $\text{list}(n'_1, n'_2)$.

We want to highlight two crucial facts of extend and append. First, it takes constant time to perform them over $\mathcal{D}$. Indeed, both operations are an addition or modification of just one node and this can be done in constant time in the RAM model. Second, although $\mathcal{D}$ is lost after applying any of the two operations (i.e., $\mathcal{D}$ is mutated into $\mathcal{D}'$), the set of complex events $\text{CE}(n_1, n_2)$ represented by $(n_1, n_2)$ in $\mathcal{D}$ are preserved in $\mathcal{D}'$. More specifically, for any pair of nodes $(n_1, n_2)$ of $\mathcal{D}$ such that $n_1$ can reach $n_2$, it holds that $\text{ENUMERATE}(\mathcal{D}, n_1, n_2, C)$ and $\text{ENUMERATE}(\mathcal{D}', n_1, n_2, C)$ give the same output. In other words, CCES is a partially persistent data structure (Driscoll et al., 1989), which means that previous versions of the structure can be accessed but only the newest version can be modified. This fact will be relevant for the update phase: we can extend the data structure with new complex events while keeping the outputs of the previous versions untouched.

**Update phase.** The last ingredient for the evaluation of CEA is the update phase. Specifically, the evaluation algorithm for an I/O-deterministic CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ over a stream $S = t_0 t_1 \ldots$ is given in Algorithm 2. Let us say that a state $q$ is active
at a certain event in the stream if there exist a run of \( \mathcal{A} \) on the prefix of \( S \) seen so far that ends in \( q \). While reading each event of \( S \), the purpose of the algorithm is to keep the set of states that are active and a CCES \( D \) to store complex events for each active state. For each active state \( q \) we maintain a pair \( (n_1, n_2) \) such that \( n_1 \) can reach \( n_2 \) and \( CE(n_1, n_2) \) is the set of all complex events produced by runs that reach \( q \). Then, for each new event \( t_i \), we update the set of active states and our data structure by using the operations \textit{extend} and \textit{append} over \( D \). After the update is done, we check for all active states that are final and output \( CE(n_1, n_2) \), that is, by using the procedure \textit{Enumerate}. 

---

**Algorithm 2** Evaluate \( \mathcal{A} = (Q, \Delta, q_0, F) \) over a stream \( S = t_0 t_1 t_2 \ldots \)

\begin{verbatim}
1: procedure EVALUATE(\( \mathcal{A}, S \))
2: \( (D, \bot) \leftarrow \text{init}() \)
3: \( T \leftarrow \emptyset \)
4: \( T[q_0] \leftarrow (\bot, \bot) \)
5: \textbf{while} \( t_i \leftarrow \text{yield}_S \) \textbf{do}
6: \( T' \leftarrow \emptyset \)
7: \textbf{for all} \( p \in \text{dom}(T) \) \textbf{do}
8: \( (n_1, n_2) \leftarrow T[p] \)
9: \textbf{if} \( q_\star \leftarrow \Delta(p, t_i, \bullet) \) \textbf{then}
10: \( (D, n) \leftarrow \text{extend}(D, i, n_1, n_2) \)
11: \( (D, T') \leftarrow \text{ADD}(D, T', q_\star, n, n) \)
12: \textbf{if} \( q_\circ \leftarrow \Delta(p, t_i, \circ) \) \textbf{then}
13: \( (D, T') \leftarrow \text{ADD}(D, T', q_\circ, n_1, n_2) \)
14: \( T \leftarrow T' \)
15: \text{OUTPUT}(T, F)
16: \textbf{procedure ADD}(D, T, q, n_1, n_2)
17: \textbf{if} \( T[q] \) is not defined \textbf{then}
18: \( T[q] \leftarrow (n_1, n_2) \)
19: \textbf{else}
20: \( (n_1', n_2') \leftarrow T[q] \)
21: \( D \leftarrow \text{append}(D, n_1', n_2, n_1) \)
22: \( T[q] \leftarrow (n_1', n_2) \)
23: \textbf{return} \( (D, T) \)
24:
25: \textbf{procedure OUTPUT}(T, F)
26: \textbf{for all} \( p \in F \cap \text{dom}(T) \) \textbf{do}
27: \( (n_1, n_2) \leftarrow T[p] \)
28: \text{ENUMERATE}(D, n_1, n_2, \emptyset)
\end{verbatim}
To remember the set of active states, we use a lookup table indexed by states that stores a pair \((n_1, n_2)\) for each state. Formally, this lookup table is a partial function \(T : Q \rightarrow \mathbb{N} \times \mathbb{N}\) for a set of nodes \(\mathbb{N}\). We write \(\text{dom}(T)\) to denote all the states that have an entry in \(T\) and \(\emptyset\) to denote the empty lookup table. Then, if \(q \in \text{dom}(T)\), we write \(T[q]\) to retrieve the pair of nodes stored for \(q\), and we say that \(T[q]\) is not defined if \(q \notin \text{dom}(T)\). We use the notation \(T[q] \leftarrow (n_1, n_2)\) to declare an update on the entry \(q\) of \(T\) with \((n_1, n_2)\). Finally, we assume that each query or update to the lookup table takes constant time, by the assumption of the RAM model.

Algorithm 2 starts by initializing a new CCES \(D\) consisting only of the special node \(\bot\), setting the lookup table \(T\) to empty, and updating the entry of the initial state \(q_0\) to \((\bot, \bot)\) (see lines 2-4). Next it reads the stream \(S\) by calling the yield method and gets the next tuple \(t_i\) where \(i\) is its position in the stream. For each \(t_i\), it builds the next lookup table \(T'\) from \(T\), starting from an empty table \(\emptyset\) (line 6). The update of \(T'\) goes by iterating over each active state \(p\) in \(T\) (i.e., the set \(\text{dom}(T)\)) and “firing” the \(\bullet\)- and \(\circ\)-transitions of \(p\) with \(t_i\). To ease the notation here, we extend \(\Delta\) as a function \(\Delta(p, t_i, m)\) that for each \(m \in \{\bullet, \circ\}\) retrieves the (unique) state \(q_m = \Delta(q, P, m)\) for some predicate \(P\) such that \(t_i \in P\); if there is no such \(P\), it returns \text{false}. With this notation, in line 9 and line 12 we fire the \(\bullet\)- and \(\circ\)-transitions of \(p\), and store its corresponding reachable state in \(q_\bullet\) and \(q_\circ\), respectively. If any of the two transitions cannot be fired (i.e. the output is false), then nothing is done.

The most crucial steps of Algorithm 2 are in lines 10-11 and line 13, namely, the update of a \(\bullet\)- and \(\circ\)-transition, respectively. First, in line 8 the pair of \(p\) is retrieved from \(T\) and instantiated in \((n_1, n_2)\). If the \(\bullet\)-transition is fired, then we must extend each complex event represented by \((n_1, n_2)\) with the new event \(t_i\) (i.e., position \(i\)). For this, we use the \texttt{extend} method of \(D\) (line 10), by getting the new version of \(D\) and the new node \(n\). The new list, represented by \((n, n)\), is added to the current list of \(q_\bullet\) in \(T'\), by calling the special method \texttt{ADD} (defined at the right column of Algorithm 2). Similarly, if the \(\circ\)-transition is fired, then we add the list \((n_1, n_2)\) directly to the list of \(q_\circ\) in \(T'\) (i.e. no extension is needed).
The method \( \text{ADD}(\mathcal{D}, T, q, n_1, n_2) \) is a common procedure to both transitions and is in charge of appending the list represented by \((n_1, n_2)\) to the list of \(q\) stored in \(T\). For this, we first need to check whether there is a list in \(T[q]\) or not, and add it directly if not (lines 17-18). Otherwise, we retrieve \(T[q]\) in \((n'_1, n'_2)\), append \((n_1, n_2)\) to \((n'_1, n'_2)\), and store the new list \((n'_1, n'_2, n_2)\) in \(T[q]\) (lines 20-22). Finally, we output the new version of \(\mathcal{D}\) and the updated lookup table \(T\).

After iterating over all states \(p \in \text{dom}(T)\), the new lookup table \(T'\) contains all the active states after reading \(t_i\). The last step is to switch \(T\) with \(T'\) (line 14) and call \(\text{OUTPUT}(T, F)\) (line 15). The \(\text{OUTPUT}\) procedure (lines 25-28) checks which of the active states are final. For such a state \(p\), it instantiates its list \(T[p]\) in \((n_1, n_2)\) and call the enumeration procedure \(\text{ENUMERATE}\) with \((n_1, n_2)\), starting with an empty complex event.

**An example.** Consider the CEA \(\mathcal{A}\) and stream \(S\) from Fig. 3.8. \(\mathcal{A}\) is in normal form, and corresponds to the CEL formula \(U; ([V + ; W) \lor W]\) (it is a simplified version of the CEA in Fig. 3.3). In particular,

\[
[A]_3(S) = \{\{0, 3\}, \{0, 1, 3\}, \{0, 2, 3\}, \{0, 1, 2, 3\}\}.
\]

The lower half of Fig. 3.8 depicts how Algorithm 2 modifies \(\mathcal{D}\) and \(T\) as it processes the stream. In particular, each subfigure jointly illustrates both \(\mathcal{D}\) and \(T\): \(\mathcal{D}\) is illustrated in...
black, while the entries of $T$ are illustrated in blue. next edges are depicted in dashed lines, first and last edges in solid lines.

The first subfigure depicts $D$ and $T$ after initialization (lines 2–4) while the others depict $D$ and $T$ after completion of the while loop (lines 6–14) on event $S[i]$, for $0 \leq i \leq 3$.

Concretely, for each new event $S[i]$ to be processed, $T'$ is initially set to empty and $T$ refers to the lookup table of the previous event. When processing $S[0]$, there are two applicable transitions: $(q_1, \text{TRUE}, \circ, q_1)$ and $(q_1, \text{tuples}(U), \bullet, q_2)$. When $(q_1, \text{TRUE}, \circ, q_1)$ is processed, line 13 copies $T[q_1] = (\bot, \bot)$ to $T'[q_1]$. When $(q_1, \text{tuples}(U), \bullet, q_2)$ is processed, line 10 calls $\text{extend}$ to create node 0 with child list $(\bot, \bot)$, and line 11 sets $T'[q_2] = (0, 0)$. Then $T$ is overwritten by $T'$ in line 14, leading to the situation as depicted in subfigure $S[0]$ of Fig. 3.8.

When processing $S[1]$, there are three applicable transitions: $(q_1, \text{TRUE}, \circ, q_1)$, $(q_2, \text{tuples}(V), \bullet, q_2)$, and $(q_2, \text{TRUE}, \circ, q_2)$. When $(q_1, \text{TRUE}, \circ, q_1)$ is processed, line 13 copies $T[q_1] = (\bot, \bot)$ to $T'[q_1]$. When $(q_2, \text{tuples}(V), \bullet, q_2)$ is processed, line 10 calls $\text{extend}$ to create node 1 with child list $(0, 0)$, and line 11 sets $T'[q_2] = (1, 1)$. When $(q_2, \text{TRUE}, \circ, q_2)$ is processed, line 13 will cause the existing list of $q_2$ in $T'$, namely $T'[q_2] = (1, 1)$ to be appended with the list of $q_2$ in $T$, namely $T[q_2] = (0, 0)$. As such, $T'[q_2]$ is set to $(1, 0)$. Then $T$ is overwritten by $T'$ in line 14, leading to the situation as depicted in subfigure $S[1]$ of Fig. 3.8.

Processing $S[2]$ and $S[3]$ proceeds similarly. In particular, when $S[3]$ is processed, final state $q_3$ becomes active. As such, line 15 causes $\text{ENUMERATE}(D, 3, 3, \emptyset)$ to be called. We invite the reader to check that this correctly enumerates $[A]_3(S)$.

**Correctness.** For an accepting run $\rho$ of $A$, let us write $\text{events}(\rho)$ for the complex event produced by $\rho$. To show the correctness of the update phase, we need to show that, after the while-loop is done (lines 6–14), then the CCES $D$ satisfies properties (1) to (5). Moreover, if $T_i$ is the $i$-th version of lookup table $T$ before processing $t_i$ (line 5) then the following two invariants must hold:
(†) For every \( p \in Q, p \in \text{dom}(T_i) \) if, and only if, \( p \) is an active state of \( \mathcal{A} \) after reading \( t_0 \ldots t_{i-1} \).

(‡) For every \( p \in \text{dom}(T_i) \) and \((n_1, n_2) = T_i[p]\), it holds that \( C \in \text{CE}(n_1, n_2) \) if, and only if, there exists a run \( \rho \) of \( \mathcal{A} \) over \( t_0 \ldots t_{i-1} \) ending in \( p \) such that \( \text{events}(\rho) = C \).

Properties (1) to (3) of the CCES \( D \) are satisfied given that \( D \) is accessed only through methods \text{extend} and \text{append}. In fact, we need to prove that the preconditions of both methods are satisfied before each call. For this, one can check by induction that \( n_1 \) can reach \( n_2 \) and \( \text{next}(n_2) = \emptyset \) for each \( p \in \text{dom}(T) \) and \((n_1, n_2) = T[p]\). This holds for the initial case \( T[q_0] \) and is preserved after each call to \text{extend} and \text{append}. Property (4) holds given that we always extend \( D \) with the last position \( i \), and Property (5) holds by the I/O determinism of \( \mathcal{A} \). Note that this is the reason why the determinism of \( \mathcal{A} \) is needed.

For each \( i \), let \( A_i \) be the set of active states of \( \mathcal{A} \) after reading the prefix \( t_0 \ldots t_{i-1} \) of \( S \). The set \( A_i \) can be recursively defined as follow: \( A_0 = \{q_0\} \) and \( A_{i+1} = \bigcup_{p \in A_i} \{ \Delta(p, t_i, \bullet), \Delta(p, t_i, \circ) \} \) for every \( i \geq 0 \). Then showing invariant (†) is the same as showing that \( A_i = \text{dom}(T_i) \) for every \( i \). This can be proved by induction over \( i \) and the recursive definition of \( A_i \).

For the last invariant (‡), consider the set \( \lbrack \mathcal{A} \rbrack_i^p(S) \) as the set of all complex events \( C \) such that there exists a run \( \rho \) of \( \mathcal{A} \) over \( t_0 \ldots t_{i-1} \) ending in \( q \) such that \( \text{events}(\rho) = C \). Similar than for \( A_i \), the set \( \lbrack \mathcal{A} \rbrack_i^p(S) \) can be recursively defined as follow. For the base case, we define \( \lbrack \mathcal{A} \rbrack_0^p(S) = \{\emptyset\} \) and \( \lbrack \mathcal{A} \rbrack_0^p(S) = \emptyset \) for every \( q \neq q_0 \). For any \( i > 0 \) and \( q \in Q \), the set \( \lbrack \mathcal{A} \rbrack_i^p(S) \) can be defined as:

\[
\lbrack \mathcal{A} \rbrack_i^p(S) = \bigcup_{\Delta(p, t_i, \bullet) = q} \{ C \cup \{i\} \mid C \in \lbrack \mathcal{A} \rbrack_{i-1}^p(S) \} \cup \bigcup_{\Delta(p, t_i, \circ) = q} \lbrack \mathcal{A} \rbrack_{i-1}^p(S)
\]

(**)

To prove (‡) we need to show that \( \text{CE}(n_1, n_2) = \lbrack \mathcal{A} \rbrack_i^p(S) \) for every \( q \in \text{dom}(T_i) \) and \((n_1, n_2) = T_i[q]\). The goal is to prove this equivalence by induction on \( i \) and using the recursive definition of \( \lbrack \mathcal{A} \rbrack_i^p(S) \). For the base case, Algorithm 2 initializes \( T_0 \) with
$T_0[q_0] = (\bot, \bot)$ and $T_0[q] = \emptyset$ for all $q \neq q_0$ (lines 3-4). Here, we can check that $\mathsf{CE}(\bot, \bot) = \{\emptyset\} = \lceil \mathcal{A}\rceil_{0}^0(S)$ and $\lceil \mathcal{A}\rceil_{i-1}^0(S) = \emptyset$ for every $q \neq q_0$. For the inductive case, note that in lines 7-13 we iterate over each $p \in \text{dom}(T_{i-1}) = A_{i-1}$ (by invariant (†)) and pick $(n_1, n_2) = T_{i-1}[p]$. Moreover, $\mathsf{CE}(n_1, n_2) = \lceil \mathcal{A}\rceil_{i-1}^p(S)$ by inductive hypothesis. Then, if $q = \Delta(p, t_i, \bullet)$, we extend each $C \in \mathsf{CE}(n_1, n_2)$ with $i$ and add this to the results in $T_i[q]$. This is equivalent to the left union of (**). Instead, if $q = \Delta(p, t_i, \circ)$, we add $\mathsf{CE}(n_1, n_2)$ directly to the results in $T_i[q]$, which is equivalent to the right union of (**). In other words, after the for-loop of lines 7-13 is done, $T_i[q] = (n'_1, n'_2)$ satisfies that $\mathsf{CE}(n'_1, n'_2) = \lceil \mathcal{A}\rceil_i^0(S)$ by equation (**). This concludes the proof for invariant (‡).

To finish the correctness of Algorithm 2, we note that for each call to yield $S$ the update procedure iterates (in the worst case) over each state $p \in Q$ and updates the data structure, where each operation takes constant time. To fire the transitions of $p$ (i.e., $\Delta(p, t_i, \bullet)$ or $\Delta(p, t_i, \circ)$), we need to iterate over all transitions of $p$ and check if $t_i$ satisfies the predicate of the transition or not, which is bounded by the function $W_U$. Furthermore, checking whether there is an output or not takes time proportional to $|Q|$. Overall, the update phase takes time proportional to $|\mathcal{A}| \cdot W_U(|t_i|)$ as expected.
Chapter 4. PARTITION-BY: THE CER CORRELATION OPERATOR

One of the key features in CER is correlation (Cugola & Margara, 2012b): to associate different events that might occur arbitrarily far in the input stream. Verifying that two users have the same id, or verifying an increasing sequence of temperature events, are some examples of how correlation is used in CER. The most basic operator for adding correlation in CER are equalities, namely, joining two events which have the same data value. Unfortunately, the evaluation of join queries is a difficult task even in a static setting (Abiteboul et al., 1995), stressing the difficulties of finding efficient evaluation algorithms of CER queries with equality predicates. One special operator usually included in CER systems (Arasu et al., 2006; Wu et al., 2006; Esper Enterprise Edition website, n.d.) for correlating events is partition-by (Arasu et al., 2006) (also referred as segmentation-oriented context in (Etzion, Niblett, & Luckham, 2011) or just context in (Esper Enterprise Edition website, n.d.)). As the name suggests, this operator breaks up the events of a stream into partitions where all events of the same partition have the same data value. Despite being a useful operator in CER, there is a lack of research in evaluating partition-by queries with solid efficiency guarantees, and usually this operator is severely restricted in CER systems (Wu et al., 2006).

In this chapter, we embark on the search for efficient evaluation of CER queries with correlation when equality and disequality predicates are used. We first formalize the partition-by operator by extending CEL with a simple and compositional semantics. To motivate the expressive power of partition-by, we show that CEL with partition-by (but without iteration) is equally expressive as hierarchical queries (Berkholz et al., 2017; Idris et al., 2017), the biggest subclass of conjunctive queries (CQ) that can be evaluated with constant update time and constant delay enumeration (Berkholz et al., 2017).

With a well-defined operator for doing correlation, we study the evaluation of partition-by through a machine model that we called chain Complex Event Automata (chain-CEA), an extension of CEA with equality and disequality predicates. Although automata models over data words usually do not have good closure properties (Segoufin,
Figure 4.1. A stream $S$ of events from Twitter. $T$ are tweets with an id, a user-id and a post message, and $R$ are responses with an id, a user-id, a tweet-id, and a reply message. The last line is the index of each event in the stream, respectively. #vote, #ihate, #stop stand for the hashtags #voteforjohn, #ihatejohn and #stophating, in the fictional scenario of US election debates on Twitter.

2006), we show that the chain-CEA model admits determinization and is expressive enough to capture all CEL queries with partition-by. The most important result of this chapter is a streaming evaluation algorithm for the full class of chain-CEA, with constant update time and output-linear delay enumeration. In particular, this shows that all queries with partition-by can be evaluated efficiently in a streaming fashion.

### 4.1. Partition-by

Our main motivation in this chapter is to study queries with correlation in CER. One of the main operators for joining multiple events is partition-by (Esper Enterprise Edition website, n.d.; Wu et al., 2006), also referred as segmentation-oriented context in (Etzion et al., 2011) or just context in (Esper Enterprise Edition website, n.d.). Intuitively, events in a stream are usually correlated by an attribute that has the same value, e.g., an id. Then this attribute is “partitioning” the stream in multiple streams, where all events of the same stream contain the same value. In this section, we formally define the PART-BY operator in CEL, and motivate its usefulness by showing that it is expressive enough to define hierarchical queries.

To better illustrate this operator, consider the following running example:

**Example 3. Consider the following two formulas**

$$\psi_1 := (T \text{ IN } X; R \text{ IN } Y) \text{ FILTER } (X.post = \texttt{'#vote'} \text{ AND } Y.reply = \texttt{'#ihate'})$$
\[ \psi_2 := (T \text{ IN } X \; ; \; (R + ) \text{ IN } Y \; ; \; R \text{ IN } Z) \text{ FILTER } (X.\text{post} = \#vote) \]

\[ \text{ AND } Y.\text{reply} = \#ihate \; \text{ AND } Z.\text{reply} = \#stop \]

In the former, suppose that a journalist wants to detect all pairs of events composed by a tweet followed by a response containing ‘#voteforjohn’ and ‘#ihatejohn’, respectively, representing “hot” debates in Twitter about the election of a candidate called John. In the latter, suppose he wants to find all sequences of debates that start with a tweet with ‘#voteforjohn’, are followed by one or more responses with ‘#ihatejohn’, and end with a response containing ‘#stophating’. For the running example, consider the stream in Figure 4.1.

Given two formulas \( \varphi_1 \) and \( \varphi_2 \), we denote by \( \varphi_1 \subseteq \varphi_2 \) when \( \varphi_1 \) is a subformula of \( \varphi_2 \). Consider a formula \( \varphi \) and variables \( X_1, \ldots, X_k \) of \( \varphi \). We say that \( X_1, \ldots, X_k \) form a variable cover of \( \varphi \) if, for every atomic subformula \( \rho \) of \( \varphi \), i.e. \( \rho \subseteq \varphi \) and \( \rho = R \) for some \( R \), there is some \( i \leq k \) and formula \( \psi = \psi' \text{ IN } X_i \) such that \( \rho \subseteq \psi \subseteq \varphi \), namely, all the events captured by atomic subformulas will be captured by some of the variables \( X_1, \ldots, X_k \) in \( \varphi \). For example, in Example 3 variables \( X, Y \) and \( Z \) form a variable cover of \( \psi_2 \).

We extend the syntax of CEL with the operator PART-BY as follows. A formula \( \varphi \) is in pCEL if it satisfies the syntax of CEL, plus the following rule:

\[ \varphi := \varphi \text{ PART-BY } [X_1.a_1, \ldots, X_k.a_k] \]

where \( X_1, \ldots, X_k \in L \) form a variable cover of \( \varphi \) and \( a_1, \ldots, a_k \in A \) are attributes. The semantics of the PART-BY operator is defined as follows. Consider a complex event \( C \), a stream \( S = t_1 t_2 \ldots \), positions \( i, j \in \mathbb{N} \) and a valuation \( \mu \). Then, \( \mu \in \llbracket \varphi \text{ PART-BY } [X_1.a_1, \ldots, X_k.a_k] \rrbracket(S, i, j) \) if \( \mu \in \llbracket \varphi \rrbracket(S, i, j) \) and for all \( m, n \leq k, l_1 \in \mu(X_m) \) and \( l_2 \in \mu(X_n) \), it holds that \( S[l_1].a_m = S[l_2].a_n \). Thus, all events must contain the same data value in their attributes. For the case we only want to partition using a single attribute \( a \) that is common among all events (e.g., an id), we add the syntactic sugar \( \varphi \text{ PART-BY } [a] \), which is defined as \( \varphi \text{ PART-BY } [a] := (\varphi \text{ IN } X) \text{ PART-BY } [X.a] \),
where $X$ is a fresh variable that does not appear in $\varphi$. Clearly, $X$ is a variable cover of $\varphi$.

**Example 4.** In Example 3 we wanted to extract all pairs of tweets and replies that contain #voteforjohn and #ihatejohn, respectively. Although $\psi_1$ extract these complex events, it fails to relate a reply with the tweet is replying to. For this, we can use the partition-by operator as follows:

$$\psi_1^* := ((T \text{ IN } X; R \text{ IN } Y) \text{ FILTER } (X.\text{post} = '\#vote' \text{ AND } Y.\text{reply} = '\#ihate')) \text{ PART-BY } (X.\text{id}, Y.\text{tweet-id})$$

Clearly, $X, Y$ form a variable cover of $\psi_1$. Furthermore, PART-BY restricts the output to pairs $t$ and $r$ with $t.\text{id} = r.\text{tweet-id}$. In Figure 4.1 now only $\{1, 2\}$, $\{1, 4\}$ and $\{5, 6\}$ are in $[\psi_1^*](S)$.

**Example 5.** Now, we want to restrict formula $\psi_2$ in Example 3 in order to correlate tweets and replies in a meaningful way. Suppose that we want to restrict $\psi_2$ such that all replies are replying to $T$ and all #ihatejohn replies are from the same user. Then we can extend $\psi_2$ with PART-BY to impose these restrictions (we omit the filters for the sake of readability):

$$\psi_2^* = [(T \text{ IN } X; (R + ) \text{ PART-BY } (\text{user-id}) \text{ IN } Y; R \text{ IN } Z) \text{ FILTER } (\cdots)] \text{ PART-BY } (X.\text{id}, Y.\text{tweet-id}, Z.\text{tweet-id})$$

This formula shows the advantage of using nesting of PART-BY. The internal PART-BY over attribute $\text{user-id}$ restricts all #ihatejohn replies to have the same identifier, namely, they come from the same user. Then the external PART-BY forces all replies to have the same $\text{tweet-id}$ as the first tweet and, therefore, they are replies of the same tweet. In Figure 4.1, $\{1, 3, 4, 6, 8\}$ is no longer an output but $\{1, 2, 4, 8\}$ still is.

### 4.1.1. Partition-by and hierarchical queries

Partition-by models a join operator that usually appears in CER systems (Arasu et al., 2006; Esper Enterprise Edition website, n.d.; Wu et al., 2006). Although this
operator can be considered rather restrictive, interestingly, it is related to the class of hierarchical queries (Dalvi & Suciu, 2007; Koutris & Suciu, 2011), the biggest class of conjunctive queries without projection that can be evaluated in a streaming fashion (Berkholz et al., 2017; Idris et al., 2017). To formally define hierarchical queries we first introduce some notation. Given a database schema $R$, we assume an arbitrary total order $<$ over the attribute names $A$. For $R \in R$ with $\text{att}(R) = \{a_1, \ldots, a_k\}$ and $a_1 < \ldots < a_k$, we write $R(x_1, \ldots, x_k)$ for variables $x_1, \ldots, x_k$ to denote that $x_i$ is assigned to attribute $a_i$. We call $R(x_1, \ldots, x_k)$ an atom. A (full) conjunctive query $Q$ is an expression $R_1(\bar{x}_1) \land \ldots \land R_k(\bar{x}_k)$ where each $R_i(\bar{x}_i)$ is an atom (i.e. we restrict our discussion to CQ without projection). Given a conjunctive query $Q$ with $k$ atoms and a stream $S = t_1t_2\ldots$ we say that a complex event $C$ satisfy $Q$ if $|C| \leq k$ and $\{t_i \mid i \in C\} \models Q$. We define $[Q]_n(S)$ as all complex events $C$ that satisfy $Q$ and $\max(C) = n$.

From now on, we restrict our analysis to hierarchical conjunctive queries. Specifically, for a variable $x$ define the set $\text{atom}(x)$ of all atoms in $Q$ where $x$ is mentioned. Then $Q$ is hierarchical (Dalvi & Suciu, 2007; Koutris & Suciu, 2011) if for every $x$ and $y$ it holds that either $\text{atom}(x) \subseteq \text{atom}(y)$, $\text{atom}(x) \supseteq \text{atom}(y)$, or $\text{atom}(x) \cap \text{atom}(y) = \emptyset$. For example, the query $R(x) \land S(x, y)$ is hierarchical and $R(x) \land S(x, y) \land T(y)$ is not.

Unfortunately, pCEL is not enough to capture the expressiveness of hierarchical queries. The reason is that partition-by combined with sequencing forces all events with correlated values to be “adjacent”. On the other hand, hierarchical queries do not impose any order over tuples. For this reason, we consider the $\text{ALL}$-operator, which we already studied in Section 3.1. Interestingly, $\text{CEL} \cup \{\text{ALL}, \text{PART-BY}\}$ captures exactly the expressiveness of hierarchical queries.

**Proposition 8.** For every hierarchical query $Q$, there exists a formula $\varphi$ in $\text{CEL} \cup \{\text{ALL}, \text{PART-BY}\}$ such that $[Q]_n(S) = [\varphi]_n(S)$ for every stream $S$ and position $n$, and vice versa.
PROOF. We prove this by showing containment in both directions. First, consider a hierarchical query \( Q \). We will define a CEL∪ \{ALL, PART-BY\} formula \( \varphi_Q \) that is equivalent to \( Q \). In (Berkholz et al., 2017) they show that there is a q-tree for every hierarchical query \( Q \), i.e., a tree \( \mathcal{T}_Q = (V, E) \) with \( V = \text{var}(Q) \) the set of variables in \( Q \) where, for each atom \( R(x_1, \ldots, x_l) \) in \( Q \), the projection of \( \mathcal{T} \) over vertices \( \{x_1, \ldots, x_l\} \) forms a directed path in \( \mathcal{T}_Q \) that starts from the root. For example, a q-tree for query \( Q' = R(x, y, z) \land S(x, y) \land T(x, w) \) is:

![Diagram of a q-tree]

Now, fix \( \mathcal{T} \) to be a q-tree for \( Q \). Let \( \text{child}(x) \) be the set of variables that \( x \) is pointing to in \( \mathcal{T} \) and \( \text{atom-end}(x) \) be the set of atoms of \( Q \) for which their path in \( \mathcal{T} \) end at \( x \). For each atom \( R(\bar{x}) \), let \( \text{attr}_R(x) \) be the attribute of \( R \) associated to variable \( x \in \bar{x} \).

We assign to every atom \( R(\bar{x}) \) a SO-variable \( X_{R(\bar{x})} \). Now, we define recursively for each variable \( x \) a formula \( \varphi_x \) in the following way:

\[
\varphi_x = ((\varphi_{y_1} \text{ ALL} \ldots \text{ ALL} \varphi_{y_l}) \text{ ALL} (R_1 \text{ IN} X_{R_1(\bar{x}_1)} \text{ ALL} \ldots \text{ ALL} R_m \text{ IN} X_{R_m(\bar{x}_m)}))
\]

\[
\text{PART-BY } (X_{T_1}.\text{attr}_{T_1}(x), \ldots, X_{T_n}.\text{attr}_{T_n}(x))
\]

where \( \{y_1, \ldots, y_l\} \) equals \( \text{child}(x) \), \( \{R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m)\} \) equals \( \text{atom-end}(x) \) and \( \{T_1(\bar{z}_1), \ldots, T_n(\bar{z}_n)\} = \text{atom}(x) \). If \( |\text{atom}(x)| = 1 \), the \text{PART-BY} can be omitted, as there is no point in partitioning on one atom. Finally, we define \( \varphi_Q = \varphi_x \) where \( x \) is the root of \( \mathcal{T} \).

For example, consider \( Q' \) and the q-tree for it shown above, consider the schema is \( R(r_1, r_2, r_3), S(s_1, s_2) \) and \( T(t_1, t_2) \), and consider the SO-variables \( X_R, X_S, X_T \) for atoms \( R(x, y, z) \), \( S(x, y) \) and \( T(x, w) \), respectively. Then,

- \( \varphi_z = R \text{ IN} X_R \),
- \( \varphi_y = (R \text{ IN} X_R \text{ ALL} S \text{ IN} S) \text{ PART-BY } (X_R.r_2, X_S.s_2) \),
• \( \varphi_w = T \text{ IN } X_T \).

Finally, the resulting formula for \( Q \) is:

\[
\varphi_x = \left( (R \text{ IN } X_R \text{ ALL } S \text{ IN } X_S) \text{ PART-BY } (X_R, r_2, X_S, s_2) \right) \text{ ALL } T \text{ IN } X_T
\]

\[
\text{PART-BY } (X_{R, r_1}, X_{S, s_1}, X_T, t_1)
\]

The correctness of the construction follows from showing, for every variable \( x \), the equivalence of \( \varphi_x \) and the subquery of \( Q \) that considers only the atoms of \( \text{atom}(x) \) and the variables in the subtree of \( T \) with root \( x \).

Now, for the opposite direction, we consider a formula \( \varphi \) of \( \text{CEL} \cup \{\text{ALL}, \text{PART-BY}\} \) and give an equivalent hierarchical query \( Q_\varphi \). For every atom \( R(\bar{x}) \) and every attribute \( a \) of \( R \), let \( \text{var}_{R(\bar{x})}(a) \) be the variable associated to attribute \( a \). The query \( Q_\varphi \) is defined recursively over the structure of \( \varphi \). In the recursion, we keep a function \( \text{var-atoms} \) that maps each variable \( X \) to the atoms it covers. The recursion goes as follows:

- If \( \varphi = R \), then \( Q_\varphi = R(\bar{x}) \) for new variables \( \bar{x} \), and \( \text{var-atoms}_\varphi = \emptyset \).
- If \( \varphi = \varphi_1 \text{ IN } X \), then \( Q_\varphi = Q_{\varphi_1} \) and \( \text{var-atoms}(X) \) is the set of all atoms in \( Q_{\varphi_1} \).
- If \( \varphi = \varphi_1 \text{ ALL } \varphi_2 \), then \( Q_\varphi = Q_{\varphi_1} \land Q_{\varphi_2} \) and, for all \( X \), \( \text{var-atoms}_\varphi(X) = \text{var-atoms}_{\varphi_1}(X) \cup \text{var-atoms}_{\varphi_2}(X) \).
- If \( \varphi = \varphi_1 \text{ PART-BY } [X_1.a_1, \ldots , X_n.a_n] \), then for each \( X_i \) let \( V_i \) be the set of all \( \text{var}_{R(\bar{x})}(a_i) \) for each atom \( R(\bar{x}) \in \text{var-atoms}_{\varphi_1}(X_i) \), and let \( V = \bigcup_i(V_i) \). Then, \( Q_\varphi \) comes from replacing all occurrences of the variables of \( V \) in \( Q_{\varphi_1} \) with a new variable \( x \).

The correctness can be proven inductively over each step of the construction. \( \square \)

The previous proposition shows the motivation of partition-by from the perspective of hierarchical CQ. Although pCEL is not enough to capture the expressibility of hierarchical CQ, it shows that partition-by is related with a subclass of CQ that can be evaluated efficiently in a streaming fashion.
4.2. Chain Complex Event Automata

Similarly to the previous chapter, we base our evaluation approach on an automata model to represent pCEL. We present an automata model, called chain Complex Event Automata (chain-CEA), and show that each formula in pCEL can be represented by this model.

In order to express the PART-BY operator, the automata model needs to be able to handle equality predicates. Given attributes \( a, b \in \mathbf{A} \) define the equality and disequality predicates as \( P_{a=b} = \{(t_1, t_2) \mid a \in \text{att}(t_1) \land b \in \text{att}(t_2) \land t_1.a = t_2.b\} \) and \( P_{a \neq b} = \text{tuples}(\mathcal{R}) \setminus P_{a=b} \). A conjunctive binary predicate, or binary predicate for short, is a predicate \( B \) that is a conjunction of equality and disequality predicates, i.e., \( B = \bigcap_{i=1}^{n} (P_{a_i=b_i}) \), where \( a_i, b_i \in \mathbf{A} \) and \( \sim_i \in \{=, \neq\} \). For simplicity, we usually drop the predicate notation and denote \( B \) simply as \( \bigwedge_{i=1}^{n} (a_i \sim b_i) \). For example, \( (a = b \land c \neq d) \) represents the predicate \( B = P_{a=b} \cap P_{c \neq d} \), and thus \( (t_1, t_2) \in B \) if \( t_1.a = t_2.b \) and, if \( c \in \text{att}(t_1) \) and \( d \in \text{att}(t_2) \), then \( t_1.c \neq t_2.d \). To separate equalities and disequalities from \( B \), we will usually denote \( B = B_{=} \land B_{\neq} \) where \( B_{=} \) and \( B_{\neq} \) are binary predicates composed only by equalities and disequalities, respectively. We denote by \( \mathbf{B} \) the set of all binary predicates.

A chain complex event automaton (chain-CEA) is a tuple \( \mathcal{A} = (Q, \Delta, I, F) \) where \( Q \) is a finite set of states, the transition relation \( \Delta \) is a set of tuples \( (p, P, B, q) \), where \( p, q \in Q, P \in \mathcal{U} \) and \( B \in \mathbf{B} \), and \( I, F \subseteq Q \) are the initial and final set of states, respectively. A configuration of \( \mathcal{A} \) is defined by a state and a position in the stream, i.e., a pair \( (q, i) \in Q \times \mathbb{N} \). An initial configuration is a pair \( (q, i) \) where \( q \in I \) and \( i = 0 \). A run \( \rho \) of \( \mathcal{A} \) over a stream \( S = t_1t_2 \ldots \) is a sequence of configurations: \( (q_0, i_0) \xrightarrow{p_1/B_1}(q_1, i_1) \xrightarrow{p_2/B_2} \ldots \xrightarrow{p_n/B_n}(q_n, i_n) \) such that \( (q_0, i_0) \) is an initial configuration and, for every \( j \leq n \): \( i_{j-1} < i_j, (q_{j-1}, P_j, B_j, q_j) \in \Delta, t_{i_j} \in P_j \) and \( (t_{i_j-1}, t_{i_j}) \in B_j \), where we consider \( t_0 \) the empty tuple with no attributes. Further, the run \( \rho \) above induces the complex event \( C_\rho = \{i_j \mid j > 0\} \). We say that \( \rho \) is an accepting run if \( q_n \in F \). We define the set of complex events of \( \mathcal{A} \) over \( S \) ending at position \( n \) as \( [\mathcal{A}]_n(S) = \{C_\rho \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ and } \max\{C\} = n\}. \)
It is worth noting that, even though only conjunctions and negations of equality predicates are allowed, in practice every logical combination (i.e. $\land$, $\lor$ and $\neg$) can be managed by simulating disjunction using multiple transitions. However, we need this restricted definition to later simplify the evaluation algorithm in Section 4.3.

**Example 6.** Recall our complex events in Example 3 of a tweet with #voteforjohn, followed of one or more responses with #ihatejohn, and ending with a response saying #stophating. Suppose now that instead of correlating all responses with the first tweet, we want to extract a chain of responses, namely, for each contiguous responses $r_1$ and $r_2$ it holds that $r_1.id = r_2.tweet-id$ (i.e. $r_2$ is a reply of $r_1$). In Figure 4.2 we show a chain-CEA defining this query. If the automaton is in the initial state $q_1$ and receives a tweet $t$ event containing #voteforjohn, it moves to $q_2$ and stores $t$. Then for each response $r$ containing #ihatejohn whose $tweet-id$ is equivalent to the id of the stored event, it forgets that event and stores $r$. Finally, when it receives an $R$-event containing #stophating which is responding the stored event, it reaches a final state.

The previous example shows a meaningful CER query definable by a chain-CEA. This type of queries are very useful in practice (see for example query (7) in (Cugola & Margara, 2012b)). The next result shows that chain-CEA is expressive enough to cover the class of pCEL formulas.

**Proposition 9.** For any formula $\varphi$ in pCEL, there exists a chain-CEA $A$ such that $[\varphi]_n(S) = [A]_n(S)$ for every $S$ and $n$.

**Proof.** We define for each $\varphi$ in pCEL a chain-CEA $A_\varphi$. We build $A_\varphi$ constructively over the structure of $\varphi$. Most of the construction is based on the construction in Theorem 1. The most important part of the construction is the one for operator.
PART-BY, which is the one explained in more detail. During the construction, we keep a function \(\text{trans}\) which, for every variable \(X\) returns a set of transitions.

- If \(\varphi = R\), then \(A_\varphi = (Q_\varphi, \Delta_\varphi, I_\varphi, F_\varphi)\) has the form:

\[
\begin{array}{c}
q_1 \\
\text{type}(R)
\end{array} \rightarrow \begin{array}{c}
q_2
\end{array}
\]

And \(\text{trans}(X) = \emptyset\) for all \(X\). Basically, \(A_\varphi\) only reads an \(R\)-tuple and retrieves its position.

- If \(\varphi = \psi \text{ IN } X\), then \(A_\varphi = A_\psi\), \(\text{trans}_\varphi(X) = \Delta\) and \(\text{trans}_\varphi(Y) = \text{trans}_\psi(Y)\) for all \(Y \neq X\).

- If \(\varphi = \psi \text{ FILTER } P(X)\), then we add predicate \(P\) to every transition in \(\text{trans}(X)\), that is, \(Q_\varphi = Q_\psi \cup \{q\}_{p, P, B, q} \in \text{trans}_\psi(X)\} \cup \{e \mid e\in \Delta_\psi \setminus \text{trans}_\varphi(X)\}\). Also, \(\text{trans}_\varphi(Y) = \text{trans}_\psi(Y)\) for all \(Y\).

- If \(\varphi = \psi_1 \text{ OR } \psi_2\), then \(A_\varphi\) is the usual union construction for automata, i.e.,

\[
Q_\varphi = Q_{\psi_1} \cup Q_{\psi_2}, \quad I_\varphi = I_{\psi_1} \cup I_{\psi_2}, \quad F_\varphi = F_{\psi_1} \cup F_{\psi_2} \quad \text{and} \quad \Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \setminus \{(p, P, B, q) \mid q \in I_{\psi_2} \land \exists q' \in F_{\psi_1}, (p, P, B, q') \in \Delta_{\psi_1}\}.
\]

Also, \(\text{trans}_\varphi(X) = \text{trans}_{\psi_1}(X) \cup \text{trans}_{\psi_2}(X)\) for all \(X\).

- If \(\varphi = a_1 \text{ PART-BY } [X_1, a_1, \ldots, X_n, a_n]\), we do the following transition-to-state construction. Define, for every transition \(e \in \Delta_\psi\), a state \(q_e\), and define \(Q_\varphi = \{q_e \mid e \in \Delta_\psi\} \cup \{q_0\}, I_\varphi = \{q_0\} \text{ and } F_\varphi = \{q_{\text{init}} \mid e = (p, P, B, q) \in \Delta_\psi \land q \in F_\psi\}, \) where \(q_0\) is a new initial state. Given two sets of variables \(\bar{X}_1 = \{X_{i_1}, \ldots, X_{i_m}\}\) and \(\bar{X}_2 = \{X_{j_1}, \ldots, X_{j_n}\}\), define the binary predicate \(B[\bar{X}_1, \bar{X}_2]\) containing all equalities between attributes of the variables in \(\bar{X}_1\) and \(\bar{X}_2\), i.e.,

\[
B[\bar{X}_1, \bar{X}_2] = \bigwedge_{k \in [1,m], l \in [1,n]} (a_{ik} = b_{il})
\]
Finally, we define the transition relation as the set $\Delta_{\psi} = \{(q_{e_1}, P, B) \land B[\text{var}(e_1) \text{var}(e_2)], q_{e_2}) \mid e_1, e_2 \in \Delta_{\psi} \land e_2 = (p, P, B, q) \} \cup \{(q_0, P, B, q_e) \mid e \in \Delta_{\psi} \land e = (p, P, B, q)p \in I_{\psi}\}$. Basically, for every pair of consecutive transitions $e_1, e_2 \in A_{\psi}$, each transition $(q_{e_1}, P, B, q_{e_2})$ in $A_{\varphi}$ simulates the transition $e_2$ of $A_{\psi}$, while also adding the equalities given by the PART-BY operator, restricted to the variables of $e_1$ and $e_2$.

The correctness of the construction can be verified by checking that the equivalence between $A_{\varphi}$ and $\varphi$ is maintained at each step of the construction. □

On the other hand, one can show that the chain-CEA from Example 6 cannot be defined by any pCEL formula. This, together with Proposition 9, shows that that pCEL is strictly included in the queries definable by chain-CEA.

4.2.1. I/O-determinization and disequalities

Like for CEA of Chapter 3, here the determinization of chain-CEA is a crucial property for having efficient streaming evaluation and necessary property for removing duplicate runs that produce the same output. We start by defining our notion of deterministic chain-CEA. Similarly to CEA, a deterministic chain-CEA must be “deterministic” with respect to the input and output, namely, given a stream $S$ and a complex event $C$, there exists at most one run over $S$ that can produces $C$. Formally, we say that a chain-CEA $A = (Q, \Delta, I, F)$ is I/O deterministic (or just deterministic) if $|I| = 1$ and, for every pair of transitions $(p, P_1, B_1, q_1) \neq (p, P_2, B_2, q_2)$, it holds that $(P_1 \cap B_1[t]) \cap (P_2 \cap B_2[t]) = \emptyset$ for every tuple $t$, where $B_i[t]$ is the set of all $t'$ such that $(t, t') \in B_i$. In other words, the conditions $(P_1, B_1)$ and $(P_2, B_2)$ must be disjoint. One can easily check that the chain-CEA from Example 6 is deterministic.

**Theorem 7.** Chain-CEA are closed under I/O-determinization, namely, for any chain-CEA $A$ there exists a I/O-deterministic chain-CEA $A'$ such that $[A]_n(S) = [A']_n(S)$ for every $S$ and $n$.

**Proof.** We give a construction that for any chain-CEA $A$ defines a deterministic chain-CEA $A_{\text{det}}$ equivalent to $A$. To simplify the presentation, we consider an
extended version of chain-CEA where the binary predicates of transitions are not restricted to conjunctions only, but they now allow any combination of boolean operations. For example, a transition $e = (p, P, B, q)$ can now have $B := a = b \lor \neg(c = b \land c = d)$. Note that this does not add any expressibility to the model, since any of these transitions can be replaced with a set of transitions with conjunctions only, by rewriting the formula in DNF and adding a transition for each clause of the disjunction. For example, $e$ above can be replaced with transitions $(p, P, (a = b), q)$ and $(p, P, (c = b \land c = d), q)$.

Consider a chain-CEA $A = (Q, \Delta, I, F)$. Let $P (B)$ be the sets of all the unary (binary, resp.) predicates of the transitions of $A$, i.e. $P = \{ P \mid (p, P, B, q) \in \Delta \}$ and $B = \{ B \mid (p, P, B, q) \in \Delta \}$. We define $A^{\text{det}} = (Q^{\text{det}}, \delta^{\text{det}}, q_0^{\text{det}}, F^{\text{det}})$ component by component. The set of states is the power set of $Q$, i.e. $Q^{\text{det}} = 2^Q$. The initial state is $q_0^{\text{det}} = I$ and the set of final states is $F^{\text{det}} = \{ S \mid S \cup F \neq \emptyset \}$.

For the transition relation $\delta^{\text{det}}$ we add some further notation. Define the equivalence relation $\equiv_P$ between tuples such that, for every pair of tuples $t_1$ and $t_2$, $t_1 \equiv_P t_2$ holds if, and only if, both satisfy the same predicates, i.e., $t_1 \in P$ holds iff $t_2 \in P$ holds, for every $P \in P$. Moreover, for every tuple $t$ let $[t]_P$ represent the equivalence class of $t$ defined by $\equiv_P$, that is, $[t]_P = \{ t' \mid t \equiv_P t' \}$. Notice that, even though there are infinitely many tuples, there is a finite number of equivalence classes which is bounded by all possible combinations of predicates in $P$, i.e., $2^{|P|}$. Now, for every $t$, define the predicate:

$$P_t = (\bigwedge_{t \in P} P) \land (\bigwedge_{t \notin P} \neg P)$$

and define the new set of predicates $\text{types}(P) = \{ P_t \mid t \in \text{tuples}(\mathcal{R}) \}$. Notice that for every tuple $t$ there is exactly one predicate in $\text{types}(P)$ that is satisfied by $t$, and that predicate is precisely $P_t$.

We define a similar idea with the binary predicates. Define the equivalence relation $\equiv_B$ between pairs of tuples such that, for every $(t_1, t_2)$, $(u_1, u_2) \equiv_B (t_1, t_2)$ holds if, and only if, $(t_1, t_2) \in B$ iff $(u_1, u_2) \in B$, for every $B \in B$. Let $[(t_1, t_2)]_B$ represent the equivalence class of $(t_1, t_2)$: $[(t_1, t_2)]_B = \{ (u_1, u_2) \mid (t_1, t_2) \equiv_B (u_1, u_2) \}$. Again, even though there are infinite pairs, the number of equivalence
classes is bounded by $2^{|B|}$, that is, by all possible combinations of predicates in $B$.

Now, for every pair $(t_1, t_2)$, define the predicate:

$$B_{t_1,t_2} = \left( \bigwedge_{(t_1, t_2) \in B} B \right) \land \left( \bigwedge_{(t_1, t_2) \notin B} \neg B \right)$$

and define the new set of predicates $\text{types}(B) = \{ B_{t_1,t_2} \mid (t_1, t_2) \in \text{tuples}(\mathcal{R})^2 \}$. For every pair $(t_1, t_2)$, $B_{t_1,t_2}$ is the only predicate in $\text{types}(B)$ that is satisfied by $(t_1, t_2)$.

Now we are ready to define the transition relation. For every state $S_1 \in Q_{\text{det}}$ and predicates $P \in \text{types}(\mathcal{P})$ and $B \in \text{types}(B)$, we add to $\Delta^{\text{det}}$ a transition $(S_1, P, B, S_2)$, where $S_2$ is defined as the maximal set that satisfies:

$$\forall q \in S_2. \exists (p, P', B', q) \in \Delta. (p \in S_2 \land P \subseteq P' \land B \subseteq B')$$

Namely, for every state of $S_2$ there must be a transition $e$ of $\mathcal{A}$ coming from a state of $S_1$ such that the conditions $P'$ and $B'$ of $e$ are implied by $P$ and $B$, respectively.

It is not hard to see that the maximal set is unique. Basically, for every two sets $T_1$ and $T_2$ that satisfy the this, the union $T_1 \cup T_2$ also satisfies it. This, together with the fact that all predicates of $\text{types}(\mathcal{P})$ are disjoint, and that all predicates of $\text{types}(B)$ are disjoint, shows that the resulting automaton is deterministic. The correctness of the construction, namely that $[\mathcal{A}]_n(S) = [\mathcal{A}^{\text{det}}]_n(S)$ for every $S$ and $n$, follows from proving that for every run in one chain-CEA there is a run in the other chain-CEA that yields the same complex event. This can be proven directly by doing induction over the length of the runs.

□

A natural question that arises from the definition of chain-CEA is whether disequalities are strictly necessary in an automata model for CER. For example, one can easily see that disequalities are not necessary for defining pCEL formulas, since the partition-by operator only requires to check that the same value is used through a contiguous subsequence of the output. In the next result, we show that disequalities are indeed necessary if we want to find an automata model that is closed under
PROPOSITION 10. There exists chain-CEA\(^=\) \(\mathcal{A}\) that such that there exists no I/O deterministic chain-CEA\(^=\) equivalent to \(\mathcal{A}\).

PROOF. Consider the chain-CEA\(^=\) \(\mathcal{A}\):

It is not hard to see that \(\mathcal{A}\) represents the CEL formula

\[
\varphi = ((R \cdot S) \text{ PART-BY } [a] \cdot T) \text{ OR } (R \cdot (S \cdot T) \text{ PART-BY } [a])
\]

i.e. that \([\mathcal{A}]_n(S) = [\varphi]_n(S)\) for every stream \(S\) and position \(n\).

By contradiction, assume there exists a deterministic chain-CEA\(^=\) \(\mathcal{A}^*\) that is equivalent to \(\mathcal{A}\). Consider, for every \(i, j\) the stream \(S_{ij} = R(i)S(j)\), namely, the stream with two events: the first one of type \(R\) with \(a\)-value \(i\), and the second one of type \(T\) with \(a\)-value \(j\). We argue that, as \(\mathcal{A}^*\) cannot use disequalities, it cannot check that \(S_{ij}\) has values \(i \neq j\) for every pair of values \(i, j\). Then, there must exist values \(i^*, j^*\) such that, while reading \(S_{i^*j^*}\) and \(S_{i^*i^*}\), \(\mathcal{A}^*\) ends at the same state, call it \(p\).

Now, consider we concatenate at the end of \(S_{i^*j^*}\) a new event \(U(k)\) for an arbitrary \(k \neq i^*\). Then, because \(\mathcal{A}\) outputs \(\{1, 2, 3\}\), there must be a run of \(\mathcal{A}^*\) that reaches an accepting state \(q\). Call \(e_k = (p, P, B, q)\) the transition it takes to reach \(q\). If we then choose some \(k^*\) such that \(k^* \neq i^*\) and \(k^* \neq j^*\), we know that \(e_{k^*}\) cannot use registers to verify the value of \(k^*\). Moreover, because \(k^*\) was not seen before, then \(e_{k^*}\) must have \(B = \text{TRUE}\) and \(U(k^*) \in P\). Finally, since \(\mathcal{A}^*\) reached the same state \(p\) reading \(S_{i^*j^*}\), then it can also take \(e_{k^*}\) reading \(U(k^*)\). Therefore when reading \(R(i^*)T(j^*)U(k^*)\) it would end in an accepting state and incorrectly output the result \(\{1, 2, 3\}\). □
We are ready to state the main result of this chapter.

**Theorem 8.** For every chain-CEA, there exists a streaming evaluation algorithm with constant update time and output-linear delay enumeration.

By combining Proposition 9 and Theorem 8, we get that for any formula in pCEL there exists a streaming evaluation algorithm with constant update time per tuple and output-linear delay enumeration. It is important to stress that chain-CEA is more general than pCEL, in particular, the chain-CEA in Figure 4.2 cannot be defined by a pCEL formula, but it can still be evaluated efficiently. We leave open whether there exists a set of operators $O$ such that CEA $∪ O$ characterizes what is definable by chain-CEA.

### 4.3. A Streaming Evaluation Algorithm for Chain-CEA

In this section we show how to evaluate a chain-CEA over a stream with constant update time and output-linear delay enumeration. We explain first the main data structures used by the algorithm to later show how to evaluate a chain-CEA.

#### 4.3.1. The run DAG.

In our algorithms, we compactly represent sets of runs by using a directed acyclic graph (DAG) annotated with configurations. Formally, let $\mathcal{A} = (Q, \Delta, q_0, F)$ be a deterministic chain-CEA. A run DAG $G$ of $\mathcal{A}$ (or just run DAG) is a tuple $G = (V, E, \bot, \kappa)$ consisting of a finite set of vertices $V$, a set of edges $E \subseteq V \times V$, a special vertex $\bot \in V$, and a function $\kappa$ that maps every $v \in V$ to a configuration $\kappa(v) \in Q \times \mathbb{N}$ of $\mathcal{A}$. It is required that the graph $(V, E)$ is acyclic, $\kappa(\bot) = (q_0, 0)$, and for every $v \in V$ there is a directed path from $v$ to $\bot$. Furthermore, it is also required that for every $(u, v) \in E$ with $\kappa(u) = (q_1, i_1)$ and $\kappa(v) = (q_2, i_2)$, it holds that $i_1 > i_2$.

Intuitively, a vertex $v$ labeled by $\kappa(v) = (q, i)$ is encoding the last configuration of a run over a stream $S$. Moreover, by the last two conditions every path starting in $v$ and ending in $\bot$ is representing a run where configurations are listed in decreasing order. We make this intuition more precise as follows. Let $\pi = v_n, \ldots, v_1, \bot$ be a path from $v = v_n$ to $\bot$ in $G$ and $\kappa(v_j) = (q_j, i_j)$ for $j \leq n$. Then $\kappa(\bot), \kappa(v_1), \ldots, \kappa(v_n)$
Algorithm 3 Enumeration of $\text{CE}(U)$

1: procedure $\text{ENUM}((U))$
2:     for all $v \in U$ do
3:         $\text{ENUMALL}(v)$
4: procedure $\text{ENUMALL}(v)$
5:     for all $v' \in \{u \mid (v, u) \in E\}$ do
6:         if $v' = \bot$ then
7:             $\text{C}.\text{enumerate}()$
8:         else
9:             $\text{C}.\text{push}(\kappa(v'))$
10:            $\text{ENUMALL}(v')$
11:        $\text{C}.\text{pop}()$

represents a run of $\mathcal{A}$ and $\text{CE}(\pi) = \{i_1, \ldots, i_n\}$ the complex event defined by $\pi$. We denote by $\text{CE}(v)$ the set of all complex events defined by paths from $v$ to $\bot$ in $G$, and $\text{CE}(U) = \bigcup_{v \in U} \text{CE}(v)$ for $U \subseteq V$.

Note that there could be two paths starting from $v$ in $G$ that define the same complex event in $\text{CE}(v)$. We say that a run DAG $G$ is safe if $\text{CE}(v_1) \cap \text{CE}(v_2) = \emptyset$ for every $v_1, v_2 \in V$. Indeed, the safety property allows to enumerate all complex events in $G$ without repetitions.

**Lemma 6.** Let $G = (V, E, \bot, \kappa)$ be a safe run DAG such that there is a procedure that, given any vertex $v \in V$, enumerates its neighborhood $\{u \mid (v, u) \in E\}$ with constant delay. Then there exists a procedure that, given $U \subseteq V$, it enumerates $\text{CE}(U)$ with output-linear delay.

**Proof.** Consider a safe run DAG $G = (V, E, \bot, \kappa)$ and that computing $\kappa(v)$ and checking $v \neq \bot$ takes constant time, and, given a vertex $v \in V$, the neighborhood $n(v) = \{u \mid (v, u) \in E\}$ can be enumerated with constant delay. We give a simple enumeration algorithm, shown in Algorithm 3 that receives as input a set $U \subseteq V$ and enumerates $\text{CE}(U)$ with output-linear delay.

Algorithm 3 uses a stack $\text{C}$ to store positions, with the typical methods: $\text{push}(i)$ to append an element $i$ at the end of $\text{C}$, and $\text{pop}()$ to remove the last element of $\text{C}$. There is also a method $\text{enumerate}$ which enumerates the current content of $\text{C}$ in linear time over the number of elements in $\text{C}$. 
Algorithm 3 is no more than a algorithm that starts from each \(v \in U\) (procedure \textsc{Enum}) and runs through all paths \(\pi\) of \(G\) from \(v\) to \(\perp\) in DFS (procedure \textsc{EnumAll}). When traversing \(\pi\), it uses \(C\) to store the complex event \(\text{CE}(\pi)\), and enumerates its content whenever it reaches \(\perp\).

It is not hard to see that Algorithm 3 enumerates \(\text{CE}(U)\) with output-linear delay. Consider that Algorithm 3 traverses all paths \(\pi_1, \pi_2, \ldots, \pi_n\) from a node of \(U\) to \(\perp\), in that order. Each run \(\pi_i\) takes at most \(|\pi_i|\) recursive calls of \textsc{EnumAll} to store \(\text{CE}(\pi_i)\) in \(C\), which takes time \(O(|\text{CE}(\pi_i)|)\). Then, enumerating \(\text{CE}(\pi_i)\) takes time \(O(|\text{CE}(\pi_i)|)\) and doing the backtracking takes time \(O(|\text{CE}(\pi_1)|)\), and then it continues with the next run \(\pi_{i+1}\). Hence, the overall time required to enumerate \(\text{CE}(\pi_i)\) is \(O(|\text{CE}(\pi_1)|)\). It is important for \(G\) to be safe, which ensures us that there are no paths of infinite length, because \(G\) is acyclic, and that each \(\pi_i\) gives a different output and, therefore, there are no repetitions. Moreover, the enumeration of the neighbourhood of a vertex is crucial to ensure that each recursive call of \textsc{EnumAll} takes constant time. \(\square\)

Therefore, by the previous lemma we can use a safe run \(\text{DAG}\) to encode the outputs of our evaluation algorithm for chain-CEA and enumerate these outputs with output-linear delay.

### 4.3.2. An index for binary predicates.

In our evaluation algorithm we will need a special index over vertices of a run \(\text{DAG}\) to efficiently evaluate the binary predicates of a chain-CEA. Given a new event \(t\) and a state \(p\), we want to quickly retrieve all configurations \((p, i)\) that have reached \(p\) and such that \((t_i, t) \in B\) for some \(e = (p, P, B, q) \in \Delta\). The run \(\text{DAG}\) will encode configurations \((p, i)\), but we will need an index to store \(t_i\) and quickly “check” \((t_i, t) \in B\).

To define this index, we first need to introduce some notation. Let \(B = \bigwedge_{i=1}^n (a_i \sim_i b_i)\) be a binary predicate with \(\sim_i \in \{=, \neq\}\). Without loss of generality, we assume that all conditions \(a_i \sim_i b_i\) in \(B\) are different. Let \(\{(a_i, b_i)\}\) be a set of fresh attributes names not used in the schema \(R\). Given a tuple \(t\), we define the left projection and right projection of \(t\) with respect to \(B\) as the tuples \(\pi_B(t)\) and \(\overrightarrow{\pi}_B(t)\), respectively,
with attributes in \( \{ (a_i, b_i) \} \) such that \( \overline{\pi}_B(t).(a_i, b_i) = t.a_i \) whenever \( a_i \in \text{att}(t) \) and \( \overline{\pi}_B(t).(a_i, b_i) = t.b_i \) whenever \( b_i \in \text{att}(t) \). Otherwise, if \( a_i \notin \text{att}(t) \) or \( b_i \notin \text{att}(t) \), then \( \overline{\pi}_B(t).(a_i, b_i) \) and \( \overline{\pi}_B(t).(a_i, b_i) \) are not defined, respectively. The left and right projections extract the relevant information of a tuple \( t \) to define \( B[t] \). To see this, we say that \( t_1 \) and \( t_2 \) are totally different, denoted by \( t_1 \neq t_2 \), if and only if \( t_1.a \neq t_2.a \) for every \( a \in \text{att}(t_1) \cap \text{att}(t_2) \), namely, they are different point-wise.

**Lemma 7.** Let \( B = B_\neq \land B_\neq \) be a binary predicate. Then \( (t, t') \in B \) if, and only if, \( \overline{\pi}_{B_\neq}(t) = \overline{\pi}_{B_\neq}(t') \) and \( \overline{\pi}_{B_\neq}(t) \neq \overline{\pi}_{B_\neq}(t') \).

**Proof.** Consider a binary predicate \( B = B_\neq \land B_\neq \), where \( B_\neq = \bigwedge_{i=1}^n (a_i = b_i) \) and \( B_\neq = \bigwedge_{j=m}^n (c_j \neq d_j) \). We prove that, for every pair of tuples \( t, t' \) it holds that \( (t, t') \in B \) if, and only if, \( \overline{\pi}_{B_\neq}(t) = \overline{\pi}_{B_\neq}(t') \) and \( \overline{\pi}_{B_\neq}(t) \neq \overline{\pi}_{B_\neq}(t') \). We prove first the only-if direction. Consider \( (t, t') \in B \). This means that \( t_a = t', b_i \) and \( t.c_j \neq t'.d_j \) for all \( i \leq n \) and \( j \leq m \). Then, by the definition of left/right projections, we get, for all \( i, j \):

\[
\overline{\pi}_{B_\neq}(t).(a_i, b_i) = t.a_i = t'.b_i = \overline{\pi}_{B_\neq}(t').(a_i, b_i) \\
\overline{\pi}_{B_\neq}(t).(c_j, d_j) = t.c_j \neq t'.d_j = \overline{\pi}_{B_\neq}(t').(c_j, d_j)
\]

Therefore, \( \overline{\pi}_{B_\neq}(t) = \overline{\pi}_{B_\neq}(t') \) and \( \overline{\pi}_{B_\neq}(t) \neq \overline{\pi}_{B_\neq}(t') \).

We now prove the if direction. Consider that \( \overline{\pi}_{B_\neq}(t) = \overline{\pi}_{B_\neq}(t') \) and \( \overline{\pi}_{B_\neq}(t) \neq \overline{\pi}_{B_\neq}(t') \). By following the argument above in the opposite direction, we get that \( (t, t') \in B_\neq \). For \( B_\neq \) we cannot do it this way directly because some attributes might not be in \( t \) or \( t' \). However, for every \( j \leq m \) such that \( c_j \notin \text{att}(t) \) or \( d_j \notin \text{att}(t') \), it still holds that \( (t, t') \in P_{c_j \neq d_j} \), and consequently \( (t, t') \in B_\neq \). Therefore, \( (t, t') \in B \) and thus the lemma holds.

With the previous notation, we are ready to define our index of a transition, called the equality-disequality index or ED-index for short. Let \( G = (V, E, \perp, \kappa) \) be a run DAG and let \( e = (p, P, B_\neq \land B_\neq, q) \) be a transition. We define the ED-index \( \text{Index}_e \) as a set of triples \( (v, t_\perp, t_\neq) \) where \( v \in V \) and \( t_\perp, t_\neq \) are left projections of \( B_\neq \) and \( B_\neq \), respectively. Intuitively, \( \text{Index}_e \) will keep all configurations that are at state \( p \) and are
“waiting” to trigger $e$. More specifically, given a stream $S = t_1 t_2 \ldots$ if $(v, t_e, t_\neq) \in \text{Index}_e$ then $\kappa(v) = (p, i)$ and $t_e = \pi_{B_e}(t_i)$ and $t_\neq = \pi_{B_\neq}(t_i)$. Thanks to Lemma 7, whenever we want to check if $(t_i, t) \in B_e \land B_\neq$ for a new tuple $t$, we only need to obtain the tuple $(v, t_e, t_\neq)$ from $\text{Index}_e$ and check whether $t_e = \pi_{B_e}(t)$ and $t_\neq \neq \pi_{B_\neq}(t)$. This motivates the following main query of an ED-index: given a pair of tuples $t'_e$ and $t'_\neq$:

$$\text{Index}_e[t'_e, t'_\neq] = \{ v \in V \mid (v, t_e, t_\neq) \in \text{Index}_e \land t_e = t'_e \land t_\neq \neq t'_\neq \}$$

That is, $\text{Index}_e[t'_e, t'_\neq]$ returns all vertices $v$ representing configurations $\kappa(v) = (p, i)$ such that there is a tuple $t'$ with $t'_e = \pi_{B_e}(t')$ and $t'_\neq = \pi_{B_\neq}(t')$ and $(t_i, t') \in B_e \land B_\neq$.

We will use the ED-index to store configurations and to quickly return them when $e$ is fired.

### 4.3.3. The streaming evaluation algorithm

In Algorithm 4 we show how to evaluate a deterministic chain-CEA over a stream. The main procedure is $\text{EVALUATION}$ that receives as input a deterministic chain-CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ and a stream $S = t_1 t_2 \ldots$. This procedure is composed by four subprocedures: $\text{INIT}$ for initializing the main data structures, $\text{FIRETRANSITIONS}(i)$ for firing the transitions in $\Delta$ given a new tuple $t_i$, $\text{UPDATEINDICES}(i)$ for updating each $\text{Index}_e$ given the previous tuple $t_i$, and, finally, $\text{ENUMERATE}$ for enumerating all complex events ending at position $i$. For the sake of presentation, instead of having a yield function that provides each next tuple in the stream, we explicitly index each new phase by $i$ (i.e. associated to tuple $t_i$) and iterate from 1 to “infinity” (the main for-loop at line 3). Then, given the next tuple $t_i$, in each $i$-phase we fire the transitions and update the indices with $t_i$, and enumerate all complex events at position $i$. In the sequel, we will first explain the data structures used by the algorithm to later describe each subprocedure.

Algorithm 4 maintains three structures that are used by all subprocedures: the run $\text{DAG} G = (V, E, \bot, \kappa)$, the ED-indices $\text{Index}_e$ for each $e \in \Delta$, and set of vertices $U_q \subseteq V$ for each $q \in Q$. As it was explained before, $G$ will encode runs of $\mathcal{A}$ and $\text{Index}_e$ will allow us to quickly evaluate the binary predicate at $e$. Moreover, for each
Algorithm 4 Evaluation of a det. chain-CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ and a stream $S = t_1t_2\ldots$

1: procedure Evaluation($\mathcal{A}$, $S$)
2: \hspace{1em} INIT()
3: for $i := 1$ to $\infty$ do
4: \hspace{2em} FireTransitions($i$)
5: \hspace{2em} UpdateIndices($i$)
6: \hspace{2em} Enumerate($\bigcup_{q \in F} U^i_q$)

7: procedure INIT()
8: $G \leftarrow$ NewMappingGraph($q_0$)
9: $U^0_{q_0} \leftarrow \{\perp\}$
10: for all $e_0 = (q_0, P, \emptyset, q) \in \Delta$ do
11: \hspace{2em} Index$^0_{e_0} \leftarrow \{(\perp, t_0, t_0)\}$

12: procedure FireTransitions($i$)
13: for all $e = (p, P, B = \land B \neq, q) \in \Delta$ do
14: \hspace{2em} $(t_\neq, t_\neq) \leftarrow (\overline{\pi}_{B_\neq}(t_i), \overline{\pi}_{B_\neq}(t_i))$
15: \hspace{2em} if $t_i \in P$ and Index$^{i-1}_e[t_\neq, t_\neq] \neq \emptyset$ then
16: \hspace{3em} $v \leftarrow$ AddNewVertex($G$, $q$, $i$)
17: \hspace{3em} Connect($G$, $v$, Index$^{i-1}_e[t_\neq, t_\neq]$)
18: \hspace{2em} $U^i_q \leftarrow U^i_q \cup \{v\}$

19: procedure UpdateIndices($i$)
20: for all $e = (p, P, B = \land B \neq, q) \in \Delta$ do
21: \hspace{2em} Index$^i_e \leftarrow$ Index$^{i-1}_e$
22: \hspace{2em} $(t_\neq, t_\neq) \leftarrow (\overline{\pi}_{B_\neq}(t_i), \overline{\pi}_{B_\neq}(t_i))$
23: for all $v \in U^i_p$ do
24: \hspace{2em} Index$^i_e \leftarrow$ Index$^i_e \cup \{(v, t_\neq, t_\neq)\}$

$q \in Q$ the set $U_q$ will keep the new vertices $v$ (i.e. configurations) at $q$. These sets will help for updating the indices and enumerating all new results. For the sake of presentation, we assume that $G$, Index$^i_e$, and $U^i_q$ are defined globally and accessible by all subprocedures.

In each $i$-phase, the algorithm will update $G$ to represent all runs of $\mathcal{A}$ over $S$ until position $i$. To that end, it will use the following methods on run DAGs. The first method, NewMappingGraph($q_0$), creates a new event DAG $G$ containing only the vertex $\perp$ with $\kappa(\perp) = (q_0, 0)$ and empty sets of vertices $V$ and edges $E$. The second method, AddNewVertex, receives an event DAG $G$ and a configuration $(q, i)$, and creates a fresh vertex $v$ with $\kappa(v) = (q, i)$, and adds it to $V$. Finally, the method returns the vertex $v$. The last method, Connect, receives as input a run DAG $G$, a
vertex \( v \) on \( G \), and a nonempty set of vertex \( U \subseteq V \), and connects \( v \) with each vertex in \( U \), namely, \((v, u)\) is added to \( E \) for every \( u \in U \). Although AddNewVertex and Connect could temporary break the properties of \( G \) (e.g. acyclicity), we will use it one after the other and it will be clear that the properties of \( G \) are always preserved.

For the structures \( \text{Index}_e \) and \( U_q \), the reader might have noticed that in Algorithm 4 we use a superscript \( \text{Index}_e^i \) and \( U_q^i \). This \( i \) is denoting the “version” of \( \text{Index}_e \) and \( U_q \) at phase \( i \). We assume that each new \( i \)-version is always initialized as empty (i.e. \( U_q^i = \emptyset \) and \( \text{Index}_e^i = \emptyset \)). It is important to note that for \( U_q^i \) we use the index \( i \) just to simplify the presentation (i.e. we could have reuse a set \( U_q \) in each phase). However, for \( \text{Index}_e^i \), the superscript is crucial to denote the version of \( \text{Index}_e \) when, for example, a vertex \( v \) is connected with the set \( \text{Index}_e^i[t_\neq, t_\neq] \) (see line 17). As it will be discussed later (see Section 4.4), \( \text{Index}_e \) is a (partially) persistent data structure (Driscoll et al., 1989) and the superscript is denoting the \( i \)-version of the structure.

We are ready to describe each subprocedure in Algorithm 4. The algorithm starts with INIT that is in charge of initializing \( G \), \( \text{Index}_e^0 \), and \( U_q^0 \) before phase 1. For this, a new event DAG \( G \) is created and the vertex with the initial configuration \( \bot \) is assigned to \( U_q^0 \) (recall that \( U_q^i = \emptyset \) for \( i \geq 0 \) by assumption). Intuitively, this represents that the initial configuration is ready to start. For initializing \( \text{Index}_e \), we assume without loss of generality that all outgoing transitions from \( q_0 \) use trivial predicates, namely, \( B = \emptyset \) for every \( e_0 = (q_0, P, B, q) \in \Delta \). Then \((\bot, t_\emptyset, t_\emptyset)\) is the only triple that must contain \( \text{Index}_e^0 \) with \( t_\emptyset \) the empty tuple.

For each new phase \( i \), we call FIRETRANSITIONS\( (i) \) that check for each transition \( e = (p, P, B_\neq \land B_\neq, q) \) whether it can be fired or not given the new tuple \( t_i \) (line 13). For this, we extract from \( t_i \) its right-projections \( t_\neq \) and \( t_\neq \) with respect to \( B_\neq \) and \( B_\neq \), respectively. Then we check if \( t_i \) satisfy \( P \) and whether there exists a previous configuration \((p, j)\) such that \((t_j, t_i)\) satisfy \( B_\neq \land B_\neq \). We do this through \( \text{Index}_e^{i-1}[t_\neq, t_\neq] \neq \emptyset \). If this is the case, all pairs of configurations \((p, j)\) and \((q, i)\) with \((p, j) \in \text{Index}_e^{i-1}[t_\neq, t_\neq] \) satisfy \( e \) and we must extend \( G \) with a new configuration \((q, i)\) that represents all these new runs. For this, we create a new node \( v \) in \( G \) for configuration \((q, i)\) and connect \( v \) with each vertex in \( \text{Index}_e^{i-1}[t_\neq, t_\neq] \)
(lines 16-17). Finally, the new vertex $v$ is added to the set $U^n_i$ of new vertices in state $q$ at phase $i$.

The next step in phase $i$ is to update $\text{Index}^{i-1}_e$ to its new version $\text{Index}_e^i$ given $t_i$. For this, we use the set $U^i_p$ to update each transition $e = (p, P, B= \land B\neq, q)$. More specifically, in $\text{UPDATEINDICES}(i)$ we iterate over each transition $e = (p, P, B= \land B\neq, q)$ and make $\text{Index}_e^i$ equal to its previous version. Then, we extract from $t_i$ its left-projections $t=\text{ }$ and $t\neq$ with respect to $B= \text{ }$ and $B\neq$, respectively, and add $(v, t=, t\neq)$ to $\text{Index}_e^i$ for each $v \in U^i_p$. Recall that $U^i_p$ contains all the new vertices added during $\text{FIRETRANSITIONS}(i)$ and, in particular, $\kappa(v) = (p, i)$ for each $v \in U^i_p$. After $\text{UPDATEINDICES}(i)$ is done, the ED-index $\text{Index}_e^i$ contains all the relevant information for checking $B= \land B\neq$ in the next phases.

Up to this point, it is straightforward to prove the following invariant after each phase $i$, which leads to the correctness proof of Algorithm 4.

**Lemma 8.** Consider $\{U_q^i\}_{q \in Q}$ and $G$ after the end of the $i$-phase. Then, for every run $(q_0, 0), (q_1, i_1), \ldots, (q_n, i_n)$ of $A$ over $S$ with $i_n = i$, there exist $v \in U^i_{q_n}$ and a path $v_n, \ldots, v_0$ in $G$ with $v_n = v$ and $v_0 = \bot$ such that $\kappa(v_j) = (q_j, i_j)$ for every $j \leq n$. Conversely, for every $v \in U^i_q$ and every path $v_n, \ldots, v_0$ in $G$ with $v_n = v$ and $v_0 = \bot$, it holds that $\kappa(v_0), \ldots, \kappa(v_n)$ is a run of $A$ over $S$. Moreover, if $A$ is deterministic, then $G$ is safe.

**Proof.** We prove in both directions. For both we prove by induction over the length $n$ (of the run and path). We begin by proving the first part. Consider a run $\rho = (q_0, 0), (q_1, i_1), \ldots, (q_n, i_n)$ of $A$ over $S$ with $i_n = i$. The base case is when $n = 0$, in which case the only run is $\rho = (q_0, 0)$ and $i = 0$. At iteration 0, the subprocedure $\text{INIT}$ added $\bot$ to $U^0_{q_0}$, and since $\kappa(\bot) = (q_0, 0)$ then the lemma holds.

For the inductive case, consider as inductive hypothesis that the first part of Lemma 8 holds for $n - 1$. Since $\rho' = (q_0, 0), (q_1, i_1), \ldots, (q_{n-1}, i_{n-1})$ is a run of $A$, then by induction hypothesis there is a path $v_{n-1}, \ldots, v_1 \bot$ in $G$ with $v_{n-1} \in U^i_{q_{n-1}}$ and $\kappa(v_j) = (q_j, i_j)$ for all $j \leq n - 1$. Consider that $e = (q_{n-1}, P, B, q_n)$ is the last transition of $\rho$, and consider the left and right projections of $t_{i_{n-1}}$ and $t_i$: $t_i^{a-1} = \ldots$
\(\overline{\pi}_{B_n}(t_{n-1})\), \(t_{n-1}^n = \overline{\pi}_{B_n}(t_{n-1})\), \(t_n^n = \overline{\pi}_{B_n}(t_n)\), \(t_{n-1}^n = \overline{\pi}_{B_n}(t_{n-1})\). Note that, at the \(i_{n-1}\)-phase, the UPDATEINDICES subprocedure added the triple \((v_{n-1}, t_{n-1}^n, t_{n-1}^n)\) to \(\text{Index}_{e}^{n-1}\) and, since we never remove elements of \(\text{Index}_{e}^{n-1}\) in the subsequent phases, \((v_{n-1}, t_{n-1}^n, t_{n-1}^n)\) is in \(\text{Index}_{e}^{n}\). Moreover, because \((t_{n-1}, t_n) \in B\), then by Lemma 7 it holds that \(t_{n-1}^n = t_n^n\) and \(t_{n-1}^n = t_{n}^n\) and by definition \(v_{n-1}\) must be in \(\text{Index}_{e}[t_n^n, t_{n}^n]\).

Then, at the \(i_n\)-phase, when performing the update for transition \(e\), the subprocedure FIRETRANSITIONS will add to \(G\) a new vertex \(v_n\) with \(\kappa(v_n) = (q_n, i_n)\), add it to \(U_{q_n}\) and connect it with all vertices of \(\text{Index}_{e}[t_n^n, t_{n}^n]\), including \(v_{n-1}\). Therefore, we proved that the path \(\pi := v_n, v_{n-1}, \ldots, v_1\downarrow\) is in \(G\) and satisfies Lemma 8.

Now, we prove the second part, again with induction over \(n\). The base case \(n = 0\) is the same as above. For the inductive step, consider that the second part of Lemma 8 holds for \(n - 1\). Consider a path \(v_n, v_{n-1}, \ldots, v_1\downarrow\) of \(G\) with \(\kappa(v_j) = (q_j, i_j)\) for all \(j \leq n\) and \(v_n \in U_{q_n}\). Consider the projections \(t_{n-1}^n, t_{n}^n, t_n^n\) and \(t_{n}^n\) defined above. Since we added the vertex \(v_n\) in the \(i_n\)-phase and connected it with \(v_{n-1}\) (with the Connect method in the FIRETRANSITIONS), there must be some transition \(e = (q_{n-1}, P, B, q_n)\) such that \(t_{i_n} \in B\) and \(t_n \in \text{Index}_{e}[t_{n-1}^n, t_n^n]\). By definition of the index, this means that \(t_{n-1}^n = t_n^n\) and \(t_{n-1}^n \neq t_n^n\) and, by Lemma 7, \((t_{n-1}, t_{n}) \in B\).

Now, considering the path \(v_{n-1}, \ldots, v_1\downarrow\), then by induction hypothesis there is a run \((q_0, 0), (q_1, i_1), \ldots, (q_{n-1}, i_{n-1})\) of \(\mathcal{A}\). Moreover, because \(t_{n-1} \in P\) and \((t_{n-1}, t_{n}) \in B\), we can extend the run with \(e\), resulting in \(\rho = (q_0, 0), (q_1, i_1), \ldots, (q_n, i_n)\), which satisfies Lemma 8.

Now we prove the last part of Lemma 8, namely that if \(\mathcal{A}\) is deterministic, then \(G\) is safe, by induction over the phase \(i\). Consider a deterministic chain-CEA \(\mathcal{A}\). In the base case, at the end of phase \(i = 0\) there is only one vertex \(\perp\), so \(G\) is trivially safe. For the inductive step, consider that \(G\) is safe at the end of phase \(i - 1\). Now in the \(i\)-phase, we focus on the FIRETRANSITIONS subprocedure, which is the one that updates \(G\). Note that every new vertex \(v\) we add has the current position \(i\), so any path starting from \(\perp\) is different to all the paths from previous iterations. Then, from the induction hypothesis, the only way for \(G\) to become unsafe is if we add two different vertices \(v_1 = (q_1, i), v_2 = (q_2, i)\) and connect both with some vertex \(v' = (q', i')\),
namely if we add \((v_1, v')\) and \((v_2, v')\) to \(E\). If this is the case, then because of the first part of Lemma 8, there are two transitions \((q'_1, P_1, B_2, q_1)\) and \((q'_2, P_2, B_2, q_2)\) that can be taken simultaneously, which contradicts the fact that \(A\) is deterministic. Therefore, this cannot happen, and \(G\) remains safe after the \(i\)-phase.

The final step at phase \(i\) is to enumerate all complex events of accepting runs. For this, we call the subprocedure \texttt{ENUMERATE} over the set of vertices \(\cup_{q \in F} U_q^i\). By Lemma 8, we know that \(G\) correctly encodes all runs of \(A\) until the \(i\)-th tuple of \(S\) and, moreover, \(G\) is safe (i.e each complex event is represented by exactly one path in \(G\)). Therefore, we can easily enumerate all complex events \([A]_i(S)\) one-by-one and without repetitions, by enumerating all paths in \(G\) starting at vertices in \(\cup_{q \in F} U_q^i\) and ending at \(\bot\).

It is only left to show that Algorithm 4 satisfies constant update time and output-linear delay enumeration. To do this, we have to dig deeper into the implementation of \Index\, which is the goal of the last section.

### 4.4. A Persistent Index Structure for Equalities and Disequalities

Fix a transition \(e = (p, P, B = \land B \neq, q)\). Let \((v_0, t_0, r_0), (v_1, t_1, r_1), \ldots\) be a sequence of triples such that \(v_i\) is a vertex and \(t_i, r_i\) are tuples for all \(i \in \mathbb{N}\). Furthermore, define \(\Index_0^e = \emptyset\) and \(\Index_i^e = \Index_{i-1}^e \cup \{(v_i, t_i, r_i)\}\). Call \((v_i, t_i, r_i)\) an insertion and \(i\) the version of \Index\.

To have constant update time and output-linear delay enumeration, \Index\ must satisfy the following properties, for every pair of tuples \(t, r\) and point in time \(i \in \mathbb{N}\):

1. every new insertion in \Index\ takes constant time, and
2. for all \(j \leq i\), \(\Index_j^e[t, r]\) can be can be enumerated with constant delay.

The last condition implies that \Index\ is a persistent data structure (Driscoll et al., 1989), namely, it preserves the previous version (i.e. \(\Index_j^e\)) of itself whenever it is modified.

We claim that, if \Index\ satisfies the above three properties, then Algorithm 4 runs with constant update time and output-linear delay enumeration. First, given that \(A\) is
fixed, then it is clear that every step of Algorithm 4 can be done in constant time, except lines 15, 17, and 24. Checking whether Index\(_e\)[t, r] \(\neq\) \(\emptyset\) (line 15) or doing an insertion in Index\(_e\) (line 24) can be done in constant time by properties (2) and (1), respectively. Furthermore, one can execute Connect\((G, v, \text{Index}^i_e[t, r])\) (line 17) in constant time if, instead of coding the graph G with adjacency lists, we represent the neighborhood of each vertex v by storing \(t, r, i\) in v and, because of (2), we can later call Index\(_e^i\)[t, r] whenever needed. Finally, from Lemma 6 we know that, if the neighborhood of each vertex from a safe run DAG can be enumerated with constant delay, then CE\((U)\) can be enumerated with output-linear delay. Given that Index\(_e^i\)[t, r] allows to enumerate the neighborhood of each vertex, then the enumeration with constant delay follows.

In the sequel, we show how to implement Index\(_e\) in order to satisfy properties (1) and (2).

**Case without disequalities.** If \(e\) does not have disequalities (i.e. \(B_\neq\) is trivial), then for every \((v, t, r) \in \text{Index}_e\), we can drop \(r\) and keep only \((v, t)\). To satisfy (1) and (2) we use a key-value index \(DS\) where keys are tuples \(t\) and each value \(DS[t]\) is a list of pairs \((u_0, i_0), \ldots, (u_n, i_n)\) where each \(u_k\) is a vertex and \(i_k\) is a "timestamp", namely, the phase when \(u_k\) was inserted. Then, for every new insertion \((u_i, t_i)\) in phase \(i\), we go to \(DS[t_i]\) and insert \((u_i, i)\) at the end of the list. Finally, for every query of the form Index\(_e^j\)[t] we can go into \(DS[t]\), jump into the pair \((u_k, i_k)\) with \(i_k = i\) and enumerate \((u_k, i_k), \ldots, (u_0, i_0)\) with constant delay. Recall that by our RAM model of computation, we can find the list \(DS[t]\) and find the pair \((u_k, i_k)\) inside \(DS[t]\) in constant time. Furthermore, by keeping \(DS[t]\) as a linked list, one can easily enumerate \((u_k, i_k), \ldots, (u_0, i_0)\) with constant delay.

**Case with disequalities.** If \(e\) includes disequalities (i.e. \(B_\neq\) is non-trivial), then we need to extend our lists \(DS[t]\) to support insertions \((v_i, t_i, s_i)\) and queries Index\(_e^i\)[t, r]. For this, extend \(DS[t]\) as a list of triples \((u_0, s_0, i_0), \ldots, (u_n, s_n, i_n)\) where \(u_k\) and \(i_k\) are as before, and \(s_k\) is the tuple for supporting disequalities. Similar to the case without disequalities, for every new insertion \((v_i, t_i, s_i)\) at phase \(i\) we go into the list \(DS[t_i]\) and insert the triple \((v_i, s_i, i)\) at the end of the list. Then for every query Index\(_e^i\)[t, r] we can jump into the list \(DS[t]\), jump into the triple \((u_k, s_k, i_k)\) with \(i_k = i\) and enumerate all
$u_l$ with $l \leq k$ such that $s_l \neq r$ (i.e. $s_l$ and $r$ are totally different). Of course, this last enumeration step cannot be done with constant delay, unless some extra bookkeeping is added to the data structure. The rest of this section is then devoted to do this.

For the sake of simplification, from now on assume that each list $\mathcal{DS}[t]$ is composed only by tuples $s_1, \ldots, s_n$. Then the problem is reduced to, given a tuple $r$ and position $i$, enumerate the set $\{s_k \mid k \leq i \land s_k \neq r\}$. Without loss of generality, assume also that all $s_1, \ldots, s_n$ have the same set of attributes $A$, i.e. $\text{att}(s_k) = A$, and define $d = |A|$. If not, complete each tuple $s_k$ with the missing attributes and a fresh value for each new attribute. For example, at the left of Figure 4.3 we give a list $s_1, \ldots, s_7$ with attributes $A = \{a, b\}$ and $d = 2$ where each column is a tuple (over integers) and each row is an attribute.

Let $\bar{a} = a_1a_2\ldots a_m$ be a sequence of non-repeating attributes of $A$, and define $\bar{A}$ to be the set of all $\bar{a}$. For each tuple $s_k$ and each $\bar{a}$, we define a tuple $s_k[\bar{a}] = s_j$ with $j < k$. Strictly speaking, $s_k[\bar{a}]$ will be a (backward) pointer from $s_k$ to $s_j$ that allows us to jump to $s_k[\bar{a}]$ in constant time. Given that our analysis is in data complexity, $|\bar{A}|$ is of constant size, so we only store a constant number of pointers in each tuple $s_k$ (although exponential in $d$). In Figure 4.3, the pointers $[a]$, $[b]$, $[ab]$, and $[ba]$ of $s_7$ are displayed with arrows.

Now, for each $s_k$ in the list $\mathcal{DS}[t] = s_1, \ldots, s_n$, the tuple $s_k[\bar{a}]$ is defined recursively as follows. First, for every attribute $a \in A$, $s_k[a]$ points to the maximum $j < k$ such that $s_k.a \neq s_j.a$. Next, for each sequence $\bar{a} = a_1a_2\ldots a_m$, $s_k[\bar{a}]$ points to the maximum $j < k$ such that, for all $1 \leq l \leq m$, $s_j.a_l \neq s_k[a_1\ldots a_{l-1}].a_l$ where $s_k[\epsilon] = s_k$ ($\epsilon$ is the empty sequence in $\bar{A}$). In the case that there is no such tuple $s_j$, then $s_k[\bar{a}]$ is not defined, which means we reached the beginning of $\mathcal{DS}[t]$.

**Example 7.** Consider the list $s_1, \ldots, s_7$ at the left of Figure 4.3 and consider tuple $s_7$. Then $s_7[a] = s_5$ is the last tuple before $s_7$ with a value different than 5, and $s_7[ab] = s_4$ is the last before $s_7$ with $s_4.a = 2 \neq 5 = s_7.a$ and $s_4.b = 4 \neq 3 = s_5.b$. Similarly, $s_7[b] = s_4$ is the last node before $s_7$ with $s_4.b = 4 \neq 3 = s_7.b$, and $s_7[ab] = s_1$ is the last before $s_7$ with $s_1.b = 4 \neq 3 = s_7.b$ and $s_1.a = 1 \neq 2 = s_4.a$. 
With the previous structure over \( s_1, \ldots, s_n \), we show how to enumerate with constant delay the set \( \{ s_k \mid k \leq i \land s_k \neq r \} \) given a tuple \( r \) and index \( i \). For this, we define a procedure \( \text{findNext}(s_k, r) \) that returns the last tuple \( s_j \) with \( j < k \) such that \( s_j \neq r \) (and false if \( s_j \) does not exist). Note that, if \( \text{findNext} \) runs in constant time, then we can enumerate the set \( \{ s_k \mid k \leq i \land s_k \neq r \} \) with constant delay: first, if \( s_i \neq r \) then we enumerate \( s_i \); then for every last node \( s_k \) we enumerated, we call \( \text{findNext}(s_k, r) \) to get the next one, until \( \text{findNext} \) returns false. For computing \( \text{findNext}(s_k, r) \), let \( s := s_{k-1} \) be the node immediately before \( s_k \) in \( \mathcal{DS}[t] \). In the first step we check if \( s[\epsilon] \) fulfills the condition, namely, if \( s \neq r \). If so, we return \( s[\epsilon] \); otherwise, there must be some attribute \( a_1 \) such that \( s[\epsilon].a_1 = r.a_1 \). In the next step we consider \( s[a_1] \) and check if \( s[a_1].a \neq r.a \) for each \( a \in \text{att}(R) \setminus \{ a_1 \} \); if so, we return \( s[a_1] \). Notice we do not need to compare \( r \) with all tuples between \( s[a_1] \) and \( s[\epsilon] \) because, by definition, each tuple \( s' \) between both satisfy \( s'.a_1 = s[\epsilon].a_1 = r.a_1 \). Furthermore, we no longer need to check the value of \( a_1 \) in \( s[a_1] \) because \( s[a_1].a_1 \neq s[\epsilon].a_1 = r.a_1 \). We repeat this procedure inductively. If we are in step \( 1 \leq m < d \) and failed in all previous steps, then for \( \bar{a} = a_1 \ldots a_m \in \bar{A} \), assume \( s[a_1 \ldots a_{l-1}].a_l = r.a_l \) for every \( l \leq m \). If \( s[\bar{a}] \neq r \), return \( s[\bar{a}] \); otherwise consider some attribute \( a_{m+1} \in A \setminus \{ a_1, \ldots, a_m \} \) such that \( s[\bar{a}].a_{m+1} = r.a_{m+1} \). Then we consider \( s[\bar{a} \cdot a_{m+1}] \) in the next step. Again, we do not need to compare \( r \) with all elements between \( s[\bar{a} \cdot a_{m+1}] \) and \( s[\bar{a}] \); each tuple \( s' \) between both satisfies \( s'.a_{m+1} = s[\bar{a}].a_{m+1} = r.a_{m+1} \). Also we do not need to compare \( s[\bar{a} \cdot a_{m+1}] \) with \( r \) on \( \{ a_1, \ldots, a_{m+1} \} \) given that, by induction, \( s[\bar{a} \cdot a_{m+1}].a_{m+1} \neq s[\bar{a}].a_{m+1} = r.a_{m+1} \) and \( s[\bar{a} \cdot a_{m+1}].a_l \neq s[a_1 \ldots a_{l-1}].a_l = r.a_l \). At some point we will find some tuple that
fulfills the conditions; in the worst-case scenario we iterate \( d \) times, in which case we are sure by definition that \( s[a_1 \ldots a_d] \) satisfies the condition or is undefined (i.e. it does not exists). All in all, the procedure takes \( O(d) \) steps, which is constant. Moreover, this procedure does not use the pointers of \( s_k \), but the ones of \( s_{k-1} \). This is an important property that we use next when we want to insert a new node in \( \mathcal{D}_S[t] \).

It is left only to show how to update \( \mathcal{D}_S[t] = s_1, \ldots, s_n \) when we read a new tuple \( s_{n+1} \). For this, we add \( s_{n+1} \) to the end of the list and define \( s_{n+1}[\bar{a}] \) for each \( \bar{a} \in \bar{A} \) in the following way. If the list is empty, then \( s_{n+1}[\bar{a}] \) is undefined for all \( \bar{a} \in \bar{A} \). Otherwise, for each \( \bar{a} = a_1 \ldots a_m \) we define \( s_{n+1}[\bar{a}] \) incrementally over the length \( m \). Suppose that, \( s_{n+1}[a_1 \ldots a_l] \) is already defined for every \( l < m \). Define the tuple \( r \) such that \( r.a_l = s_{n+1}[a_1 \ldots a_{l-1}].a_l \) for all \( l < m \). Then, define \( s_{n+1}[a_1 \ldots a_m] := \text{findNext}(s_{n+1}, r) \). In other words, we collect all values \( c_1 = s_{n+1}[a].a_1 \), \( c_2 = s_{n+1}[a_1].a_2 \), \ldots, \( c_m = s_{n+1}[a_1 \ldots a_m].a_m \) and find the last tuple \( s \) such that \( s.a_l \neq c_l \) for every \( l \leq m \). As it was mentioned above, since \( \text{findNext} \) only uses the pointers of \( s_n \), and not of \( s_{n+1} \) itself, the function is well-defined. Moreover, given that \( \text{findNext}(s_{n+1}, r) \) can be found in constant time, then \( s_{n+1}[a_1 \ldots a_m] \) is computed in constant time as well.

**Example 8.** Suppose that we want to add the node \( s_8 = \{a \rightarrow 2, b \rightarrow 3\} \) to the list on the left of Figure 4.3. The result is shown on the right of Figure 4.3 where \( s_8 \) is the last dashed column. We define \( s_8[\bar{a}] \) incrementally using \( \text{findNext} \). For \( a \), we call \( \text{findNext}(n_8, \{a \rightarrow 2\}) \), which tries with the last tuple \( s_7 \) and, because \( s_7.a \neq 2 \), we set \( s_8[a] := s_7 \). For \( b \), we call \( \text{findNext}(s_8, \{b \rightarrow 3\}) \), which first tries with \( s_7 \), but \( s_7.b = 3 \), so it tries with \( s_7[b] = s_4 \); since \( s_4.b \neq s_7.b \), we set \( s_8[b] = s_4 \). For sequence \( ab \), we have \( s_8.a = 2 \) and \( s_8[a].b = 3 \), so we call \( \text{findNext}(s_8, \{a \rightarrow 2, b \rightarrow 3\}) \). As \( s_7 \) conflicts in \( b \), it tries with \( s_7[b] = s_4 \), but this time it conflicts with \( a \), so it tries with \( s_7[ba] = s_4 \). As \( s_1.a \neq 2 \) and \( s_1.b \neq 3 \), we set \( s_8[ab] = s_1 \). The same procedure is done for \( ba \), resulting in \( s_8[ba] = s_1 \).

By combining the key-value index \( \mathcal{D}_S \) where the keys are tuples and the values are the extended list with the additional bookkeeping mentioned above, we get properties
(1) and (2) needed for Algorithm 4 to have constant update time and output-linear delay enumeration.
In this chapter we abstract beyond the world of CER query evaluation to a more expressive framework: MSO. This allows for the obtained results regarding query evaluation to be applicable in other contexts, most notably in the area of information extraction.

The goal in information extraction is to extract some subparts of a text. A logical approach that has brought a lot of attention in the database community is document spanners (Fagin et al., 2015). This logical framework provides a language for extracting subparts of a document. More specifically, regular spanners are based on regular expressions that fill relations with tuples of the texts’ subparts. These relations can afterwards be queried by conjunctive or datalog-like queries.

The document spanners’ main algorithmic problem is the efficient evaluation of a spanner over a word. Recently, the most studied approach has been to focus on the enumeration problem to obtain efficient evaluation algorithms. The principle of an enumeration algorithm is to create a representation of the set of answers efficiently depending only on the input word’s size and the query, and not in the number of answers. This time is called the preprocessing time. The second part of an enumeration algorithm is to enumerate the outputs one by one using the previous representation, and the delay is the time taken between two consecutive outputs. As for the preprocessing time, an efficient delay should not depend on the number of outputs, but only on the input size (i.e., word and query). In general, the most efficient enumeration algorithms have linear preprocessing time and constant delay, both in data complexity (i.e. in the size of the input word).

The reader might already see that there is a relation between this principle of enumeration and the one we have used in previous chapters. Indeed, both share the notion of delay in output enumeration but, while so far we have measured the time to process the input as the update time, i.e., the time required to read each element of the stream and update the underlying data structure accordingly, from now on we will focus on the overall preprocessing time, i.e., the time required to consume the
complete input and build the structure. In principle, the preprocessing time approach is more restrictive because it requires the complete input to be read before enumerating the outputs, while the updating approach can enumerate the results found while the input is being read. In this chapter we follow the preprocessing approach so that the results stated here are in line with previous research in the field. However, we dedicate Section 5.3.1 to show how the following results can be applied to CER in a streaming fashion.

Several people have studied the enumeration problem over words following different formalisms. For example, (Bagan, 2006; Courcelle, 2009; Segoufin, 2013) studied the enumeration problem for MSO logic and (Florenzano et al., 2020; Amarilli et al., 2019) for regular spanners (i.e., automata). For all these formalisms, it is shown that there exists an enumeration algorithm with linear time preprocessing and delay constant both in data complexity (i.e. in the size of the input word).

The interest of an efficient enumeration algorithm is to provide a process that can quickly give the first answers. Unfortunately, these answers may not be relevant for the user; that is, the enumeration process does not assume how the output will be ordered. A classical manner of considering the user’s preferences is to associate a score to each solution and then rank them following this score; for instance one could ask for the matches ranked by order of length, or by the number of times a second pattern appears within the match. This approach of “scoring” solutions has been used particularly in the context of information extraction. Indeed, there have been several recent proposals (Doleschal et al., 2020, 2021) to extend document spanners with annotations from a semi-ring. The proposed annotations are typically useful to capture the confidence of each solution (Doleschal et al., 2020). For instance, (Doleschal et al., 2021) proves that the enumeration of the answers following their scores’ order is possible with polynomial-time preprocessing and polynomial delay.

In this chapter, we are interested in establishing a framework for scoring outputs and improve the bounds proved in (Doleschal et al., 2020). We propose using what we called MSO cost functions, which are formulas in weighted logics (Droste & Gastin, 2005) extended with open variables. These formulas provide a simple formalism for
defining the output and scoring with MSO logic. We show that one can translate each MSO cost function to a cost transducer. These machines are a restricted form of weighted functional vs-automaton (Doleschal et al., 2020), for which there exists at most one run for any word and any valuation. We use cost transducers to study the ranked enumeration problem: enumerate all outputs in increasing rank order. Specifically, the main result of the chapter is an algorithm for enumerating all the solutions of a cost transducer in increasing order efficiently; specifically, with a preprocessing phase linear in the input word and a logarithmic delay between solutions.

The preprocessing part builds a heap containing the answers with their score, and one step of the enumeration is simply a pop of the heap. For this, we use a general data structure that we called Heap of Words (HoW), having the classical heap operations of finding/deleting the minimal element, adding an element, and melding two heaps. We also need to add two new operations that allow us to concatenate a letter to and increase the score of all elements of the heap. Finally, we require that this structure is fully-persistent (Driscoll et al., 1989), i.e., that each of the previous operations returns a new heap without changing the previous one. To obtain the required efficiency, we rely on a classical persistent data structure called Brodal queue that we extend in order to capture the new operations over the stored words and scores presented above. We call this extension an incremental Brodal queue.

Finally, for ranked query evaluation, there has been recent progress in the context of conjunctive queries: on the efficient computation of top-\(k\) queries (Tziavelis, Gatterbauer, & Riedewald, 2020) and the efficient ranked enumeration (Tziavelis, Ajwani, et al., 2020; Deep & Koutris, 2019). These advances consider relational data (which is more general than words) and conjunctive queries (which is more restricted than MSO queries); they are thus incomparable to our work. However, it is important to note some similarities with our work, such as the need for an “advanced” priority queue (the Fibonacci heap (Deep & Koutris, 2019)), which means that our incremental queues might be of great interest there.

The main results of this chapter are threefold: (i) we introduce MSO cost functions, a framework to express MSO queries and scores, generalizing the proposals of
document spanners; (ii) we give a ranked enumeration scheme that has linear preprocessing time and logarithmic delay in data complexity with a polynomial combined complexity; (iii) we introduce two new data structures for our scheme: the Heaps of Words and the incremental Brodal queues. Both of these structures might be of interest in other ranked enumerations schemes.

5.1. Preliminaries

Since we will abstract from the framework of CER to the one of MSO, the first thing we need to do is formalize the setting we will work on.

5.1.1. Notation

Words. We denote by \( \Sigma \) a finite alphabet, \( \Sigma^* \) all words over \( \Sigma \), and \( \epsilon \) the empty word of 0 length. Give a word \( w = a_1 \ldots a_n \), we write \( w[i] = a_i \). For two words \( u, v \in \Sigma^* \) we write \( u \cdot v \) as the concatenation of \( u \) and \( v \). We denote by \([n] = \{1, \ldots, n\}\).

Ordered groups. A group is a pair \((G, \oplus, O)\) where \( G \) is a set of elements, \( \oplus \) is a binary operation over \( G \) that is associative, \( O \in G \) is a neutral element for \( \oplus \) (i.e., \( O \oplus g = g \oplus O = g \)) and every \( g \in G \) has an inverse with respect to \( \oplus \) (i.e., \( g \oplus g^{-1} = O \) for some \( g^{-1} \in G \)). A group is abelian if, in addition, \( \oplus \) is commutative (i.e., \( g_1 \oplus g_2 = g_2 \oplus g_1 \)). From now on, we assume that all groups are abelian. We say that \((G, \oplus, O, \preceq)\) is an ordered group if \((G, \oplus, O)\) is a group and \( \preceq \) is a total order over \( G \) that respects \( \oplus \), namely, if \( g_1 \preceq g_2 \) then \( g_1 \oplus g \preceq g_2 \oplus g \) for every \( g, g_1, g_2 \in G \). Examples of (abelian) ordered groups are \((\mathbb{Z}, +, 0, \leq)\) and \((\mathbb{Z}_k, +, (0, \ldots, 0), \leq_k)\) where \( \leq_k \) represents the lexicographic order over \( \mathbb{Z}_k \).

MSO. We use monadic second-order logic for defining properties over words. As usual, we encode words as logical structures with an order predicate and unary predicates to represent the order and the letters of each positions of the word, respectively. More formally, fix an alphabet \( \Sigma \) and let \( w \in \Sigma^* \) be a word of length \( n \). We encode \( w \) as a structure \( ([n], \leq, (P_a)_{a \in \Sigma}) \) where \([n] \) is the domain, \( \leq \) is the total order over \([n]\), and \( P_a = \{ i \mid w[i] = a \} \). By some abuse of notation, we also use \( w \) to denote its corresponding logical structure.
A MSO-formula $\varphi$ over $\Sigma$ is given by:

$$
\varphi := x \leq y \mid P_a(x) \mid x \in X \mid \varphi \land \varphi \mid \neg \varphi \mid \exists x. \varphi \mid \exists X. \varphi
$$

where $a \in \Sigma$, $x$ and $y$ are first-order (FO) variables, and $X$ is a monadic second order (MSO) variable (i.e., a set variable). We write $\varphi(\bar{x}, \bar{X})$ where $\bar{x}$ and $\bar{X}$ are the sets of free FO and MSO variables of $\varphi$, respectively. An assignment $\sigma$ for $w$ is a function $\sigma : \bar{x} \cup \bar{X} \rightarrow 2^{[n]}$ such that $|\sigma(x)| = 1$ for every $x \in \bar{x}$ (note that we treat FO variables as a special case of MSO variables). Note that assignments $\sigma$ fulfill the same need as valuations $\mu$ in CER: to map variables to sets of positions. We introduce this new notation to keep both settings self-contained.

As usual, we denote by $\text{dom}(\sigma) = \bar{x} \cup \bar{X}$ the domain of the function $\sigma$. Then we write $(w, \sigma) \models \varphi$ when $\sigma$ is an assignment over $w$, $\text{dom}(\sigma) = \bar{x} \cup \bar{X}$, and $w$ satisfies $\varphi(\bar{x}, \bar{X})$ when each variable in $\bar{x} \cup \bar{X}$ is instantiated by $\sigma$ (see (Libkin, 2013)). Given a formula $\varphi(\bar{x}, \bar{X})$, we define $[\varphi](w) = \{\sigma \mid (w, \sigma) \models \varphi(\bar{x}, \bar{X})\}$. For the sake of simplification, from now on we will only use $\bar{X}$ to denote the free variables of $\varphi(\bar{X})$ and use $X \in \bar{X}$ for an FO or MSO variable.

For any assignment $\sigma$ over $w$, we define the support of $\sigma$, denoted by $\text{sup}(\sigma)$, as the set of positions mentioned in $\sigma$; formally, $\text{sup}(\sigma) = \{i \mid \exists v \in \text{dom}(\sigma), i \in \sigma(v)\}$. Furthermore, we encode assignments as sequences over the support as follows. Let $\text{sup}(\sigma) = \{i_1, \ldots, i_m\}$ such that $i_j < i_{j+1}$ for every $j < m$. Then we define the (word) encoding of $\sigma$ as:

$$
\text{enc}(\sigma) = (\bar{X}_1, i_1)(\bar{X}_2, i_2)\ldots(\bar{X}_m, i_m)
$$

such that $\bar{X}_j = \{X \in \text{dom}(\sigma) \mid i_j \in \sigma(X)\}$ for every $j \leq m$. That is, we represent $\sigma$ as an increasing sequence of positions, where each position is labeled with the variables of $\sigma$ where it belongs. This is the standard encoding used to represent assignments for running algorithms regarding MSO formulas (Bagan, 2006; Courcelle, 2009). Finally, we define the size of $\sigma$ as $|\text{enc}(\sigma)| = |\text{dom}(\sigma)| \cdot m$. 


5.1.2. Efficiency

Given a formula \( \varphi \) and a word \( w \), the main goal of this chapter is to study the enumeration of assignments in \( [\varphi](w) \). We assume that group elements can be stored in constant space and that all group-related operations (i.e., to evaluate \( g_1 \oplus g_2 \), \( g_1 \preceq g_2 \) and \( g^{-1} \)) take constant time.

We say that an algorithm \( \mathcal{E} \) is an enumeration algorithm for MSO evaluation with \( f \)-preprocessing time and \( g \)-delay if \( \mathcal{E} \) runs in two phases, for every MSO-formula \( \varphi \) and a word \( w \).

(i) In first phase, called the **preprocessing phase**, the algorithm reads the whole input and builds a data structure \( D \), spending time bounded by \( O(f(|\varphi|)) \).

(ii) In the second phase, called the **enumeration phase**, the algorithm: (1) writes \( \# \text{enc}(\sigma_1)\# \text{enc}(\sigma_2)\# \ldots \# \text{enc}(\sigma_k)\# \) to the output registers where \( \# \) is a distinct separator symbol, and \( \sigma_1, \ldots, \sigma_k \) is an enumeration (without repetition) of the assignments of \( [\varphi](w) \); (2) it writes the first \( \# \) as soon as the enumeration phase starts; and (3) it stops immediately after writing the last \( \# \).

We measure the delay in the same way as it is described in Section 2.3. Recall that the time taken between writing the \( i \)-th \( \# \) and \( (i+1) \)-th \( \# \) is at most the size of the \( i \)-th output \( \text{enc}(\sigma_1) \) times \( g(n) \), and is \( g(n) \) if \( [\varphi]_n(S) \) is empty, up to a constant factor.

It is important to notice that, although we fix a particular encoding for assignments and we restrict the enumeration algorithms to this encoding, we can use any encoding for the assignments whenever there exists a linear transformation between \( \text{enc}(\cdot) \) and the new encoding. Given the definition of delay, if we use an encoding \( \text{enc}'(\sigma) \) for \( \sigma \), and there exists a linear time transformation between \( \text{enc}(\sigma) \) and \( \text{enc}'(\sigma) \) for every \( \sigma \), then the same enumeration algorithm works for \( \text{enc}'(\cdot) \). In particular, whenever the encoding depends linearly over \( \text{sup}(\sigma) \) and \( |\bar{\bar{x}} \cup \bar{\bar{X}}| \), then the aforementioned property holds.
5.1.3. Ranked enumeration

For an MSO formula $\varphi$ and $w \in \Sigma^*$, we consider the ranked enumeration of the set $[\varphi](w)$. For this, we need to assign an order to the outputs and we do this by mapping each element to a total order set. Fix a set $C$ with a total order $\preceq$ over $C$. A cost function is any partial function $\kappa$ that maps words $w \in \Sigma^*$ and assignments $\sigma$ to elements in $C$. Without loss of generality, we assume that $\kappa$ is defined only over pairs $(w, \sigma)$ such that $\sigma$ is an assignment over $w$.

Let $\varphi$ be an MSO formula and $\kappa$ a cost function over $(C, \preceq)$. We define the ranked enumeration problem of $(\varphi, \kappa)$ as

<table>
<thead>
<tr>
<th>Problem: RANK-ENUM[$\varphi, \kappa$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: A word $w \in \Sigma^*$</td>
</tr>
<tr>
<td>Output: Enumerate all $\sigma_1, \ldots, \sigma_k \in <a href="w">\varphi</a>$ without repetitions and such that $\kappa(w, \sigma_i) \preceq \kappa(w, \sigma_{i+1})$.</td>
</tr>
</tbody>
</table>

Note that we consider the version of the problem in data-complexity where $\varphi$ and $\kappa$ are fixed. We say that RANK-ENUM[$\varphi, \kappa$] can be solved with preprocessing time $f(n)$ and delay $g(n)$ if there exists an enumeration algorithm $E$ that runs with preprocessing time $f(n)$ and delay $g(n)$ and, for every $w \in \Sigma^*$, $E$ enumerates $[\varphi](w)$ in increasing ordered according to $\kappa$. In the next section, we give a language to define cost functions and we state our main result.

5.2. MSO Cost Functions

To state our main result about ranked enumeration of MSO, first we need to choose a formalism to define cost functions. We do this by staying in the same setting of MSO logic by considering weighted logics over words (Droste & Gastin, 2005; Droste, Kuich, & Vogler, 2009; Kreutzer & Riveros, 2013). Functions defined by extensions of MSO has been studied by using weighted automata, but also people have found it counterparts by extending MSO with a semiring. We use here a fragment of weighted MSO parametrized by an ordered group to fit our purpose.
Fix an ordered group \((G, \oplus, O, \preceq)\). A weighted MSO-formula \(\alpha\) over \(\Sigma\) and \(G\) is given by the following syntax:

\[
\alpha := [\varphi \mapsto g] \mid \alpha \oplus \alpha' \mid \Sigma x. \alpha
\]

where \(\varphi\) is an MSO-formula, \(g \in G\), and \(x\) is an FO variable. Further, we assume that the \(\Sigma x\) quantifier cannot be nested. For example, \((\Sigma x. [\varphi \mapsto g]) \oplus (\Sigma y. [\varphi' \mapsto g'])\) is a valid formula but \(\Sigma x. \Sigma y. [\varphi \mapsto g]\) is not. Similar than for MSO formulas, we write \(\alpha(\bar{x}, \bar{X})\) to state explicitly the sets of FO-variables \(\bar{x}\) and of MSO variables \(\bar{X}\) that are free in \(\alpha\).

Let \(\sigma\) be an assignment. For any FO-variable \(x\) and \(i \in \mathbb{N}\) we denote by \(\sigma[x \rightarrow i]\) the extension of \(\sigma\) with \(x\) assigned to \(i\), namely, \(\text{dom}(\sigma[x \rightarrow i]) = \{x\} \cup \text{dom}(\sigma)\) such that \(\sigma[x \rightarrow i](x) = \{i\}\) and \(\sigma[x \rightarrow i](y) = \sigma(y)\) for every \(y \in \text{dom}(\sigma) \setminus \{x\}\).

We define the semantics of a weighted MSO formula \(\alpha\) as a function from words and assignments to elements in \(G\). Formally, for every \(w \in \Sigma\) and every assignment \(\sigma\) over \(w\) we define the output \([\alpha](w, \sigma)\) recursively as follows:

\[
[[\varphi \mapsto g]](w, \sigma) = \begin{cases} g & \text{if } (w, \sigma) \models \varphi \\ O & \text{otherwise.} \end{cases}
\]

\[
[[\alpha \oplus \alpha']] (w, \sigma) = [[\alpha]](w, \sigma) \oplus [[\alpha]](w, \sigma)
\]

\[
[[\Sigma x. \alpha]](w, \sigma) = \bigoplus_{i=1}^{|w|} [[\alpha]](w, \sigma[x \rightarrow i])
\]

where \(\varphi\) is any MSO-formula, \(\alpha\) and \(\alpha'\) are weighted MSO formulas, and \(g \in G\). By some abuse of notation, in the following we will not make distinction between \(\alpha\) and \([\alpha]\), that is, the cost function over \(G\) defined by \(\alpha\).

**Example 9.** Consider the alphabet \(\{a, b\}\) and suppose that we want to define a cost function that counts the number of \(a\)-letters between two variables \(x\) and \(y\). This can be defined in weighted MSO over \(\mathbb{Z}\) as follows:

\[
\alpha_1 := \Sigma z. [x \leq z \land z \leq y \land P_a(z)] \mapsto 1
\]
Here, $\alpha_1$ uses $z$ to count over all positions of the word and we count 1 whenever $z$ is labeled with $a$ and is between $x$ and $y$, and we count 0, otherwise, which is the identity of $\mathbb{Z}$.

**Example 10.** Consider again the alphabet $\{a, b\}$ and suppose that we want a cost function to compare assignments over variables $(x, y)$ lexicographically. For this, we can write a weighted MSO-formula over $\mathbb{Z}^2$ that maps each assignment $\sigma$ over $x$ and $y$ to a pair $(\sigma(x), \sigma(y))$. This can be defined in weighted MSO over $\mathbb{Z}^2$ as follows:

$$\alpha_2 := (\sum z_1. [(z_1 \leq x) \rightarrow (1, 0)]) + (\sum z_2. [(z_2 \leq y) \rightarrow (0, 1)])$$

Similar than for the previous example, we use the $\Sigma$-quantifier to add in the first and second component the value of $x$ and $y$, respectively. In fact, for every assignment $\sigma = \{x \rightarrow i, y \rightarrow j\}$ over $w \in \Sigma^*$ it holds that $J_{\alpha_2}(w, \sigma) = (i, j)$.

Strictly speaking, the syntax and semantics of weighted MSO defined above is a restricted version of weighted logics (Droste & Gastin, 2005), in the sense that weighted logics is usually defined over a semiring, which has two binary operations $\oplus$ and $\odot$. Indeed, it will be interesting to have a better understanding of the expressibility of MSO cost functions, or to extend our results for weighted logics over semiring. We leave this for future work.

We are ready to state the main result of this chapter about ranked enumeration of MSO.

**Theorem 9.** Fix an alphabet $\Sigma$ and an ordered group $\mathbb{G}$. For every MSO formula $\varphi$ over $\Sigma$ and for every weighted MSO formula $\alpha$ over $\Sigma$ and $\mathbb{G}$, the problem $\text{RANK-ENUM}[\varphi, \alpha]$ can be solved with linear preprocessing time and logarithmic delay.

As it is common for MSO logic over words, we prove this result by developing an enumeration algorithm using automata theory. Specifically, we define a weighted automata model, that we called cost transducer, and show that its expressiveness is equivalent to the combination of (boolean) MSO and weighted MSO logic.
From now on, fix an input alphabet $\Sigma$ and an output alphabet $\Gamma$. Furthermore, fix an ordered group $(G, \oplus, O, \preceq)$. A cost transducer over $G$ is a tuple $T = (Q, \Delta, \kappa, I, F)$, where $Q$ is the set of states, $\Delta \subseteq Q \times \Sigma \times 2^I \times Q$ is the transition relation, $\kappa : \Delta \to G$ is a function that associates a cost to every transition of $\Delta$, and $I : Q \to G$, $F : Q \to G$ are partial functions that associate a cost in $G$ to (some) states in $Q$. The functions $I$ and $F$ are partial functions because they naturally define the set of initial and final states as $\text{dom}(I)$ and $\text{dom}(F)$, respectively. A run of $T$ over a word $w = a_1a_2 \ldots a_n$ is a sequence of transitions $\rho : q_0 \xrightarrow{a_1/\bar{X}_{i_1}} q_1 \xrightarrow{a_2/\bar{X}_{i_2}} \ldots \xrightarrow{a_n/\bar{X}_{i_m}} q_n$ such that $q_0 \in \text{dom}(I)$ and $(q_{i-1}, a_i, \bar{X}_i, q_i) \in \Delta$ for every $i \leq n$. We say that $\rho$ is accepting if $q_n \in \text{dom}(F)$.

For a run $\rho$ as defined above, let $\{i_1, \ldots, i_m\} \subseteq [n]$ be all the positions of $\rho$ such that $\bar{X}_{i_j} \neq \emptyset$ and $i_j < i_{j+1}$ for all $j \leq m$. Then we define the output of $\rho$ as the sequence:

$$\text{out}(\rho) = (\bar{X}_{i_1}, i_1)(\bar{X}_{i_2}, i_2) \ldots (\bar{X}_{i_m}, i_m)$$

Moreover, we extend $\kappa$ over accepting runs $\rho$ by adding the costs of all transitions of $\rho$ plus the initial and final cost, namely:

$$\kappa(\rho) = I(q_0) \oplus \bigoplus_{i=1}^{\lfloor w \rfloor} \kappa((q_{i-1}, a_i, \bar{X}_i, q_i)) \oplus F(q_n).$$

Note that $\text{out}(\rho)$ defines the encoding of some assignment $\sigma$ over $w$ with $\text{dom}(\sigma) = \Gamma$ and $\text{out}(\rho) = \text{enc}(\sigma)$. Of course, the opposite direction is not true: for some assignment $\sigma$ there could be no run $\rho$ that defines $\sigma$ and, moreover, there could be two runs $\rho_1$ and $\rho_2$ such that $\text{out}(\rho_1) = \text{out}(\rho_2) = \text{enc}(\sigma)$, but $\kappa(\rho_1) \neq \kappa(\rho_2)$. For this reason, we impose an additional restriction to cost transducers: we assume that all cost transducers in this chapter are unambiguous, that is, for every $w \in \Sigma^*$ there does not exist two distinct runs $\rho_1$ and $\rho_2$ of $w$ such that $\text{out}(\rho_1) = \text{out}(\rho_2)$. In other words, a cost transducers satisfies that for every $w \in \Sigma^*$ and assignment $\sigma$ there exists at most one run $\rho$ such that $\text{out}(\rho) = \text{enc}(\sigma)$.

Given the unambiguous restriction of cost transducers, we can define a partial function from pairs $(w, \sigma)$ to $G$ as $\text{cost}_T(w, \sigma) = \kappa(\rho)$ whenever there exists a run $\rho$ of $w$ such that $\text{out}(\rho) = \text{enc}(\sigma)$. Otherwise $\text{cost}_T(w, \sigma)$ is not defined. Given
that for some pairs \((w, \sigma)\) the function \(\text{cost}_T\) is not defined, we can define the set \([\mathcal{T}](w) = \{\sigma \mid \text{cost}_T(w, \sigma) \text{ is defined}\}\) of all outputs of \(T\) over \(w\).

It is important to notice that, given \(w \in \Sigma^*\), a cost transducer \(T\) is in charge of (1) defining the set of assignments \([\mathcal{T}](w)\) and (2) assigning a cost \([\mathcal{T}](w, \sigma)\) for each output \(\sigma \in [\mathcal{T}](w)\). These two tasks are separated in our setting of ranked MSO enumeration by having a MSO formula \(\varphi\) that defines the outputs \([\varphi]\) and a weighted MSO formula \(\alpha\) to assign a cost to each pair \((w, \sigma)\). In fact, one can show that cost transducers are equally expressive than combining MSO plus weighted MSO.

**Proposition 11.** For every cost transducer \(T\), there exists a MSO formula \(\varphi_T\) and weighted MSO formula \(\alpha_T\) such that \([\mathcal{T}](w, \sigma) = \text{cost}_T(w, \sigma)\) for every \(\sigma \in [\mathcal{T}](w)\). Moreover, for every MSO formula \(\varphi\) and weighted MSO formula \(\alpha\), there exists a cost transducer \(T_{\varphi, \alpha}\) such that \([\varphi] = [\mathcal{T}_{\varphi, \alpha}](w)\) and \([\alpha](w, \sigma) = \text{cost}_{\mathcal{T}_{\varphi, \alpha}}(w, \sigma)\) for every \(\sigma \in [\varphi](w)\).

**Proof.** The following is a proof taken from Theorems 4.1 and 5.3 of (Kreutzer & Riveros, 2013) and adapted to our setting. Instead of citing the results there, we decided to add the complete proof to keep it self-contained. First, let fix an ordered group \((\mathbb{G}, \oplus, O, \preceq)\).

\((\Rightarrow)\) Consider a cost transducer \(T = (Q, \Delta, \kappa, I, F)\) and let \(V\) be the output alphabet. We define formulas \(\varphi_T\) and \(\alpha_T\) with free variables \(V\), where the former encodes a run of \(T\) and the latter calculates the cost of such run. In the following, let \(\Delta_I\) be the set of transitions coming from an initial state and \(\Delta_F\) be the set of transitions going to a final state. Moreover, for every \(U \in V\), define the set \(\Delta_U = \{(p, a, V', q) \in \Delta \mid U \in V'\}\) of transitions that output \(U\).

We begin by defining \(\varphi_T\). First of all, we introduce some auxiliary formulas:

- \(\text{first}(x) := \forall y. x \leq y\); \(\text{last}(x) := \forall y. y \leq x\);
- \(\text{succ}(x, y) := x \leq y \land y \nless x \land \forall z. (z \leq x \lor y \leq z)\).

These denote the first element, last element and successor relation, respectively, according to the order \(\leq\) of the word structure. We encode a run \(\rho\) by defining, for each \(t = (p, a, V, q) \in \Delta\), a variable \(X_t\) such that \(i \in X_t\) if the \(i\)-th transition of \(\rho\) is \(t\). Let \(\bar{X} = \{X_t \mid t \in \Delta\}\). Now, we define the predicate \(\text{run}(\bar{X}, V)\)
that is satisfied only when $\bar{X}$ defines an accepting run in $\mathcal{T}$ such that its output defines an encoding equal to the assignment of $V$.

$$
\text{run}(\bar{X}, V) := \forall x. \bigvee_{t \in \Delta} (x \in X_t \land \bigwedge_{t' \neq t} x \notin X_{t'}) \land \\
\bigwedge_{(p,a,U,q) \in \Delta} \forall x. (x \in X_{(p,a,U,q)} \rightarrow x \in P_a) \land \\
\forall x. \forall y. \left( \text{succ}(x, y) \rightarrow \bigvee_{(p,a,U,q), (q,b,W,r) \in \Delta} x \in X_{(p,a,U,q)} \land y \in X_{(q,b,W,r)} \right) \land \\
\exists x. \left( \text{first}(x) \land \bigvee_{t \in T_I} x \in X_t \right) \land \exists x. \left( \text{last}(x) \land \bigvee_{t \in T_F} x \in X_t \right) \\
\bigwedge_{U \in V} \left( \forall x. x \in U \leftrightarrow \bigvee_{t \in \Delta_U} x \in X_t \right)
$$

The first line makes sure that exactly one transition is used per position; the second makes sure that each transition is taken only if the input letter is correct; the third checks that the transitions form a path; the fourth checks the initial and final states and the last checks that the assignment output by the run corresponds to the assignment of $V$.

Then, we define our first formula:

$$
\varphi_T(V) := \exists \bar{X}. \text{run}(\bar{X}, V)
$$

Note that $\varphi_T$ guesses a run and, because $\mathcal{T}$ is unambiguous, that run is unique for every assignment of $V$. Then it holds that, for every word $w$, $[\varphi](w) = [\mathcal{T}](w)$.

Now, to define $\alpha_T$, we define cost formulas that capture the costs of the initial state, the transitions and the final states:

$$
\text{init}(V) := \bigoplus_{(p,a,U,q) \in T_I} \left( \exists \bar{X}. \left( \text{run}(\bar{X}, V) \land \exists x. (\text{first}(x) \land x \in X_{(p,a,U,q)}) \right) \mapsto I(p) \right)
$$

$$
\text{trans}(x, V) := \bigoplus_{(p,a,U,q) \in T} \left( \exists \bar{X}. \left( \text{run}(\bar{X}, V) \land x \in X_{(p,a,U,q)} \right) \mapsto \kappa((p, a, U, q)) \right)
$$

$$
\text{final}(V) := \bigoplus_{(p,a,U,q) \in T_F} \left( \exists \bar{X}. \left( \text{run}(\bar{X}, V) \land \exists x. (\text{last}(x) \land x \in X_{(p,a,U,q)}) \right) \mapsto F(q) \right)
$$
The \textit{trans} formula is supposed to receive a position \( x \) of the domain and retrieve the cost of the \( x \)-th transition of the guessed run. Then, we define our second formula:

\[
\alpha_\text{\textit{trans}}(V) := \text{init}(V) \oplus \Sigma x. \text{trans}(x, V) \oplus \text{final}(V)
\]

It is then easy to check that \( \text{cost}_{\alpha_\text{\textit{trans}}}(w, \sigma) = \llbracket \alpha_\text{\textit{trans}} \rrbracket (w, \sigma) \) for every word \( w \) and \textit{V-assignment} \( \sigma \).

\((\Leftarrow)\) Consider an MSO formula \( \varphi \) and a weighted MSO formula \( \alpha \). Given an alphabet \( \Sigma \) and a set of (second and first order) variables \( V \), let \( \Sigma_V = \Sigma \times 2^V \) be the alphabet used to encode words and assignments. Moreover, by abuse of notation, let \( (w, \sigma) \in \Sigma_V \) be the word encoding \( w \) and \( \sigma \), i.e., \( (w, \sigma) = (a_1, V_1)(a_2, V_2) \ldots (a_n, V_n) \), where \( a_1 \ldots a_n = w \) and \( X \in V_i \) iff \( i \in \sigma(X) \). The following Lemma is a well-known result that relates the expressiveness of MSO and DFA.

**Lemma 9.** For every MSO formula \( \psi \) over \( \Sigma \) with free variables \( V \) there exists a DFA \( A_\psi \) over \( \Sigma_V \) such that for every word \( w \in \Sigma^* \) and assignment \( \sigma \) we have:

\[
(w, \sigma) \vDash \psi(V) \text{ iff } (w, \sigma) \in \mathcal{L}(A_\psi)
\]

where \( \mathcal{L}(A) \) is the set of words accepted by the DFA \( A \).

Therefore, regarding the MSO formula \( \varphi \), because of Lemma 9 we know there exists a DFA \( A_\varphi \) over \( \Sigma_V \) such that \( (w, \sigma) \vDash \varphi(V) \) iff \( (w, \sigma) \in \mathcal{L}(A_\varphi) \). Next, we build a cost transducer \( T_\alpha \) for \( \alpha \), then make the cross-product of it with \( A_\varphi \) and obtain the final cost transducer.

In particular, the cost transducer \( T_\alpha = (Q_\alpha, \Delta_\alpha, \kappa_\alpha, I_\alpha, F_\alpha) \) is equivalent to \( \alpha \) in the sense that it satisfies \( \llbracket \alpha \rrbracket (w, \sigma) = \text{cost}_{T_\alpha}(w, \sigma) \) for every word \( w \) and assignment \( \sigma \). Moreover, if all subformulas of \( \alpha \) are of the form \( \varphi \mapsto g \) or \( \alpha_1 \oplus \alpha_2 \), then \( T_\alpha \) is deterministic, meaning that it has a single accepting state \( q_0 \in \text{dom}(I_\alpha) \) and, for every \( p \in Q_\alpha, a \in \Sigma \) and \( V' \subseteq V \) there is at most one transition \( (p, a, V', q) \in \Delta_\alpha \). Usually, we use notation \( T = (Q, \delta, \kappa, I, F) \) to denote that \( T \) is deterministic. Otherwise, if \( \alpha \) also contains operation \( \Sigma \cdot x \), then \( T_\alpha \) is unambiguous.
We build $\mathcal{T}_\alpha$ by induction over the structure of $\alpha$. The base case is when $\alpha = [\varphi' \mapsto g]$, with $\varphi'$ being an MSO formula and $g \in \mathbb{G}$. Consider the DFA $A_{\varphi'} = (Q', \delta', q_0', F')$ given by Lemma 9. Then, we build $\mathcal{T}_\alpha = (Q', \delta_\alpha, \kappa_\alpha, I_\alpha, F_\alpha)$ such that:

- $\delta_\alpha = \{(p, a, V', q) \mid (p, (a, V'), q) \in \delta'\}$,
- $\kappa_\alpha(t) = O$ for every $t \in \delta_\alpha$,
- $I_\alpha(q_0') = O$, and
- $F_\alpha(q) = g$ for every $q \in F'$.

It is not hard to see that $\mathcal{T}$ is deterministic and that for every word $w$ and assignment $\sigma$, $\llbracket \alpha \rrbracket(w, \sigma) = \text{cost}_{\mathcal{T}_\alpha}(w, \sigma)$.

Now, we consider the case where $\alpha = \alpha_1 \oplus \alpha_2$. By induction, consider that there are cost transducers $\mathcal{T}_i = (Q_i, \Delta_i, \kappa_i, I_i, F_i)$ equivalent to $\alpha_i$, for $i \in \{1, 2\}$. Then, we build $\mathcal{T}_\alpha = (Q_\alpha, \Delta_\alpha, \kappa_\alpha, I_\alpha, F_\alpha)$ as the cross product of $\mathcal{T}_1$ and $\mathcal{T}_2$ in the following way:

- $Q_\alpha = Q_1 \times Q_2$,
- $\Delta_\alpha = \{(p_1, p_2), a, V', (q_1, q_2)) \mid (p_1, a, V', q_1) \in \Delta_1 \wedge (p_2, a, V', q_2) \in \Delta_2\}$,
- $\kappa_\alpha(((p_1, p_2), a, V', (q_1, q_2))) = \kappa_1(p_1, a, V', q_1) \oplus \kappa_2(p_2, a, V', q_2)$,
- $I_\alpha((q_1, q_2)) = O$ for all $q_1 \in \text{dom}(I_1)$ and $q_2 \in \text{dom}(I_2)$, and
- $F_\alpha((q_1, q_2)) = F_1(q_1) \oplus F_2(q_2)$ for all $q_1 \in \text{dom}(F_1)$ and $q_2 \in \text{dom}(F_2)$.

Again, for every word $w$ and assignment $\sigma$, $\llbracket \alpha \rrbracket(w, \sigma) = \text{cost}_{\mathcal{T}_\alpha}(w, \sigma)$. Moreover, if $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic (unambiguous), then $\mathcal{T}_\alpha$ is deterministic (unambiguous, respectively).

Finally, we consider the case where $\alpha = \Sigma x. \alpha'$. Let $\mathcal{T}' = (Q', \delta', \kappa', I', F')$ be the cost transducer equivalent to $\alpha'$. Because there are no nested $\Sigma x$ operators, we know that $\mathcal{T}'$ is deterministic. Moreover, because of the construction, for every run $\rho$ ending in some state $q_f \in \text{dom}(F')$, all costs are added by the final state, i.e., $\kappa'(t) = O$ for all $t \in \delta'$, $I(q) = O$ for all $q \in \text{dom}(I)$, and $\kappa(\rho) = F(q_f)$. We make use of this property in the following construction. For simplicity, assume that $V = \{x\}$, so that transitions
of $\mathcal{T}'$ either have the form $(p, a, \{x\}, q)$ or $(p, a, \emptyset, q)$: we call the latter $\emptyset$-transitions.

Next, we define $\mathcal{T}_\alpha = (Q_\alpha, \Delta_\alpha, \kappa_\alpha, I_\alpha, F_\alpha)$ equivalent to $\alpha$:

- $Q_\alpha = Q' \times 2Q' \times Q'$,
- $I_\alpha(q, \emptyset) = 0$ for all $q \in \text{dom}(I')$
- $F_\alpha(q, R) = 0$ if $R$ has the form $R = \{(q_1, q_1), \ldots, (q_k, q_k)\}$,
- $t = ((q_1, R_1), a, \emptyset, (q_2, R_2)) \in \Delta_\alpha$ and $\kappa(t) = s$ if there exists $q_f \in Q'$ such that
  - $F'(q_f) = s$
  - $(q_1, a, \emptyset, q_2) \in \delta'$, and
  - $R_2 = \{(p_2, q_f'_{j}) \in Q^2 \mid \exists (p_1, q_f') \in R_1, (p_1, a, \emptyset, p_2) \in \delta'\} \cup \{(q, q_g) \mid (q_1, a, \{x\}, q) \in \delta'\}$.

The motivation behind this construction is that, for each $(q, R) \in Q_\alpha$, $q$ keeps the state of the (only) run of $\mathcal{T}'$ considering that $x$ has not been assigned, while $R$ keeps pairs $(p, q_f)$ such that $p$ follows a run that has already assigned $x$ and $q_f$ is the “guessed” final state of such run. Moreover, every time a final state is guessed, its value $F(q_f)$ is added to the cost of the run. The guess then is verified by checking that each final state is reached as expected.

To check that $\mathcal{T}_\alpha$ is unambiguous, consider by contradiction that there are two accepting runs

$$\rho_i = (q_0, \emptyset) \xrightarrow{a_1/\emptyset}(q_1, T_1^{i}) \xrightarrow{a_2/\emptyset} \cdots \xrightarrow{a_n/\emptyset}(q_0, T_n^{i})$$

for $i \in \{1, 2\}$. Then, let $j \leq n$ be the smallest position at which both runs differ, namely $T_j^1 \neq T_j^2$ and $T_{j-1}^1 = T_{j-1}^2 = T_{j-1}'$. By looking at the definition of $T_j^1$ and $T_j^2$, we can see that both have the same set in the first part, defined completely by $T_{j-1}'$, which is $S = \{(p_2, q_f) \in Q^2 \mid \exists (p_1, q_f') \in T_{j-1}'(p_1, a, \emptyset, p_2) \in \delta'\}$. For the second part, let $q_f^1$ and $q_f^2$ be the final states guessed by the transitions in $\rho_1$ and $\rho_2$. Then, for $i \in \{0, 1\}$, $T_i = S \cup \{(q, q_f') \mid (q_{j-1}, a_j, \{x\}, q) \in \delta'\}$. Because $\mathcal{T}'$ is deterministic, the only way for the sets to differ is if $q_f^1 \neq q_f^2$. Now, let $p \in \mathcal{T}'$ be the state reached from $q$ after reading the remaining suffix of $w$, i.e., $w[j + 1] \ldots w[n]$, using only $\emptyset$-transitions. From the construction, $(p, q_f^1)$ must be in $T_n^1$ and $(p, q_f^2)$ must be in $T_n^2$. 

Moreover, because \( \rho_1 \) and \( \rho_2 \) are accepting, it must hold that \( q_i^1 = p = q_j^2 \), reaching a contradiction.

Now, consider a word \( w = a_1 \ldots a_n \) and the only accepting run

\[
\rho = (q_0, \emptyset) \xrightarrow{a_1/\emptyset} (q_1, T_1) \xrightarrow{a_2/\emptyset} \cdots \xrightarrow{a_n/\emptyset} (q_n, T_n)
\]

where each \( s_i \) is the cost of the \( i \)-th transition. It is not hard to see that, for every \( i \in [n] \), \( s_i = \llbracket \alpha' \rrbracket (w, \sigma[x \rightarrow i]) \) and, therefore,

\[
\llbracket \alpha \rrbracket (w, \sigma) = \sum_{i=1}^{n} s_i = \text{cost}_T(w, \sigma)
\]

Now that we have a CT \( T = (Q, \Delta, \kappa, I, F) \) and a deterministic finite-state automaton \( A = (Q, \delta, \kappa, q_0, F) \) (over words in \( \Sigma \)), we build our final CT \( T = (Q, \Delta, \kappa, I, F) \) by doing the cross-product of both as follows:

- \( Q = Q_\alpha \times Q_\phi \)
- \( \Delta = \{(p_1, p_2), a, V', (q_1, q_2)\} \mid (p_1, a, V', q_1) \in \Delta_\alpha \wedge (p_2, (a, V'), q_2) \in \delta_\phi \)
- \( \kappa((p_1, p_2), a, V', (q_1, q_2)) = \kappa_\alpha((p_1, a, V', q_1)) \)
- \( I(q_1, q_0) = I_\alpha(q_1) \) for all \( q_1 \in \text{dom}(I_\alpha) \)
- \( F(q_1, q_2) = F_\alpha(q_1) \) for all \( (q_1, q_2) \in \text{dom}(F_\alpha) \times F_\phi \)

From the above, one can check that \( \llbracket \phi \rrbracket = \llbracket T \rrbracket \) and \( \llbracket \alpha \rrbracket (w, \sigma) = \text{cost}_T(w, \sigma) \) for every word \( w \) and assignment \( \sigma \in \llbracket \phi \rrbracket (w) \).

By the previous result, we can represent pairs of formulas \((\phi, \alpha)\) by using cost transducers and vice-versa. Similar than for MSO (Reinhardt, 2002), there exists a non-elementary blow-up for going from \((\phi, \alpha)\) to a cost transducer and this blow-up cannot be avoided (Frick & Grohe, 2004).

To solve the problem \( \text{RANK-ENUM}_{[\phi, \alpha]} \) we can use a cost transducer \( T_{\phi, \alpha} \) to enumerate all its outputs following the cost assigned by this machine. More concretely, we study the following rank enumeration problem for cost transducers:
Problem: RANK-ENUM-T

Input: A cost transducer \( T \) and a word \( w \in \Sigma^* \).

Output: Enumerate all \( \sigma_1, \ldots, \sigma_k \in \mathcal{L}(w) \) without repetitions and such that \( \text{cost}_T(w, \sigma_i) \preceq \text{cost}_T(w, \sigma_{i+1}) \).

Note that for RANK-ENUM-T we consider the cost transducer as part of the input\(^1\). Indeed, for this case we can provide an enumeration algorithm with stronger guarantees regarding the preprocessing time in terms of \( T \). We now give the theorem formalizing the main result of this chapter, which will be proven in the next section:

**Theorem 10.** *The problem RANK-ENUM-T can be solved with \(|T| \cdot |w|\) preprocessing time and \(\log(|T| \cdot |w|)\)-delay.*

In the rest of the chapter, we present the above mentioned ranked enumeration algorithm. We start by showing a general algorithm based on a novel data structure called a Heap of Words. In Section 5.5, we provide the implementation of this structure. In Section 5.6, we show how to implement the incremental Brodal queues, a technical data structure needed to obtain the required efficiency. Before presenting the technical details of this algorithm, in the next section we show two applications of this result.

### 5.3. Applications

In this section we present two applications of the previous result: the first one in the framework of CER and the second one in the framework of document spanners.

#### 5.3.1. Complex Event Recognition

In this section we return to the world of CER.

To define complex events over streams, we use the model of complex event automata presented in Section 3.2 which we showed that is expressible enough to define all (regular) complex event queries over streams. However, we need to introduce a new

---

\(^1\)In Section 5.1 we introduce the setting of ranked enumeration for MSO formulas and cost functions. One can easily extend this setting and the definiton of enumeration algorithms for cost transducer.
definition for CEA: we say that \( \mathcal{A} \) is unambiguous if for every stream \( S \), position \( n \), and \( C \in [\mathcal{A}]_n(S) \), there exists exactly one run \( \rho \) of \( \mathcal{A} \) over \( S \) of length \( n \) such that \( C = \mathcal{C}_\rho \). Note that every I/O-deterministic CEA is also unambiguous, so we know that for every CEA we can build an equivalent one that is unambiguous.

Recall that, given a CEA \( \mathcal{A} \) and a stream \( S \), the evaluation problem consists in enumerating \([\mathcal{A}]_n(S)\) at each position \( n \). Of course, ranked enumeration can also be applied in this setting. More specifically, for any stream \( S = t_1 t_2 \ldots \) and for each position \( n \) we can define a structure \( S_n = ([n], \leq, (P_S)_{P \in \mathcal{U}}) \) such that \( P_S(i) \) is true if, and only if, \( t_i \in P \). To define the cost of each complex event at position \( n \), we can use a weighted MSO formula \( \alpha(X) \) over \( \mathcal{U} \) such that the cost of \( C \in [\mathcal{A}]_n(S) \) is equal to \( [\alpha](S_n, \sigma[C]) \) where \( \sigma[C] = \sigma[X \mapsto C] \). Then, given a weighted MSO formula \( \alpha(X) \) over \( \mathcal{U} \) we can state the ranked enumeration problem of CEA as follows.

\[
\begin{align*}
\text{Problem:} & \quad \text{RANK-ENUM-CEA}[\alpha] \\
\text{Input:} & \quad \text{A CEA } \mathcal{A} \text{ and a stream } S = t_1 t_2 \ldots \\
\text{Output:} & \quad \text{After reading } t_1 \ldots t_n, \text{enumerate all } C_1, \ldots, C_k \in [\mathcal{A}]_n(S) \text{ without repetitions and such that } [\alpha](S_n, \sigma[C_i]) \preceq [\alpha](S_n, \sigma[C_{i+1}]).
\end{align*}
\]

**Example 11.** A complex event \( C \) can be placed arbitrarily far from the current position \( n \) in the stream, which might give \( C \) less importance. Hence, a useful cost function is to measure the distance between the first event in \( C \) and the current position \( n \). We can define this distance with a weighted MSO formula over \( \mathbb{Z} \) as follows:

\[
\alpha_3 := \sum z.[(\exists x. x \in X \land x \leq z) \mapsto 1]
\]

One can check that \( [\alpha_3](S_n, \sigma[X \mapsto C]) = n - \min(C) + 1 \) for every stream \( S \), position \( n \), and complex event \( C \). Thus, if we enumerate \([\mathcal{A}]_n(S)\) ranked by \( \alpha_3 \), the complex events that are “closer” to \( n \) in the stream will be enumerated first.

It is important to notice that the existence of an enumeration algorithm for the ranked enumeration problem of MSO does not imply the existence of a streaming enumeration algorithm. However, as shown in Section 5.4, our ranked enumeration...
algorithm maintains a data structure to process the input in such a way that it is updated “one letter at a time” and, at any moment of the preprocessing phase, all outputs until that moment can be efficiently enumerated. That is, we can derive the following result:

**Theorem 11.** The problem RANK-ENUM-CEA[α] can be solved with $2^{|A|} \cdot |t|$ update-time and $\log(|A| \cdot |S_n|)$-delay enumeration. Furthermore, if $A$ is unambiguous, then RANK-ENUM-CEA[α] can be solved with $|A| \cdot |t|$ update-time and $\log(|A| \cdot |S_n|)$-delay enumeration.

For this result, the preprocessing part of Algorithm 4 has to be modified to call the enumeration procedure after reading each event in the stream, instead of only at the end.

An important property of streams in CER is that the relevance of events and complex events rapidly decays over time. Arguably, a user would prefer the complex event $C_1 = \{5, 6, 8, 9\}$ over $C_2 = \{1, 6, 8, 9\}$ because its time interval is shorter and closer to the current time 9. For this reason, CER query languages usually include time operators that filter out the complex events that are not inside a sliding window (Cugola & Margara, 2012b). Formally, given a CEA $A$ and a number $T$ (encoded in binary) consider the query $Q := A \text{ WITHIN } T$ such that, for every stream $S$ and position $n$, $Q$ defines the set of complex events $[A \text{ WITHIN } T]_n(S) = \{C \in [A]_n(S) \mid n - \min(C) \leq T\}$. In other words, it considers only those $C$ captured by $A$ that are contained within the last $T$ events of the stream. Interestingly, we can use Theorem 11 to efficiently evaluate queries of the form $A \text{ WITHIN } T$. Indeed, if we evaluate $A$ over a stream $S$, rank each complex event with the cost function $\alpha_3$ of Example 11, and enumerate all complex events in $[A \text{ WITHIN } T]_n(S)$. Thus, we easily get the following corollary for the evaluation of CER queries over sliding windows.

**Corollary 6.** For each CEA $A$ and value $T$ (in binary), $A \text{ WITHIN } T$ can be evaluated with $2^{|A|} \cdot |t|$ update-time and $\log(|A| \cdot |S_n|)$-delay. Furthermore, if $A$ is restricted to be unambiguous, then it can be evaluated with $|A| \cdot |t|$ update-time and $\log(|A| \cdot |S_n|)$-delay.
It is important to clarify two facts about the previous result. First, the result can be extended to CER queries over sliding windows when time is continuous (e.g. in seconds) by slightly modifying our evaluation algorithm. Second, the advantage of the previous result is that the evaluation process does not depend on the length $T$ of the sliding window. Although the length $T$ reduces the number of events that a query needs to “see” for the evaluation, given that $T$ is in binary (or time is continuous), the sliding window could contain a huge number of events during the evaluation process.

5.3.2. Document Spanners

The framework of document spanners was proposed in (Fagin et al., 2015) as a formalization of rule-based information extraction and has attracted a lot of attention both in terms of the formalism (Freydenberger, Kimelfeld, & Peterfreund, 2018; Maturana, Riveros, & Vrgoc, 2018) and the enumeration problem associated to it (Florenzano et al., 2020). Recently, an extension of document spanners has been proposed to enhance the extraction process with annotations (Doleschal et al., 2020, 2021). These annotations serve as auxiliary information of the extracted data such as confidence, support, or confidentiality measures. To extend spanners, this framework follows the approach of provenance semiring by annotating the output with elements from a semiring and propagating the annotations by using the semiring operators. Next we give the core definitions of (Doleschal et al., 2020) and we state the application of our main results to this setting.

We start by defining the central elements of document spanners: documents and spans. Fix a finite alphabet $\Sigma$. A document over $\Sigma$ (or just a document) is a string $d = a_1 \ldots a_n \in \Sigma^*$ and a span is a pair $s = [i, j)$ with $1 \leq i \leq j \leq n + 1$. A span represents a continuous region of $d$, whose content is the substring from positions $i$ to $j - 1$. Formally, the content of span $[i, j)$ is defined as $d[i, j) = a_i \ldots a_{j-1}$; if $i = j$, then $d[i, i) = \epsilon$. Fix a finite set of variables $X$. A mapping $\nu$ over $d$ is a function from $X$ to the spans of $d$. A document spanner (or just spanner) is a function that maps each document $d$ to a set of mappings over $d$. 
To annotate mappings, we need to introduce semirings. A semiring \((K, \oplus, \odot, 0, 1)\) is an algebraic structure where \(K\) is a non-empty set, \(\oplus\) and \(\odot\) are binary operations over \(K\), and \(0, 1 \in K\). Furthermore, \(\oplus\) and \(\odot\) are associative, \(0\) and \(1\) are the identities of \(\oplus\) and \(\odot\) respectively, \(\oplus\) is commutative, \(\odot\) distributes over \(\oplus\), and \(0\) annihilates \(K\) (i.e., \(0 \odot k = k \odot 0 = 0\) for all \(k \in K\)). We will use \(\bigoplus_X\) or \(\bigodot_X\) for the \(\oplus\)- or \(\odot\)-iteration over all elements in some set \(X\), respectively. An ordered semiring \((K, \oplus, \odot, 0, 1, \preceq)\) is a semiring extended with a total order \(\preceq\) over \(K\) such that \(\preceq\) preserves \(\oplus\) and \(\odot\), namely, \(k_1 \preceq k_2\) implies \(k_1 \ast k \preceq k_2 \ast k\) for \(\ast \in \{\oplus, \odot\}\). From now on, we will assume that all semirings are ordered. A semifield (Golan, 2013) is a semiring \((K, \oplus, \odot, 0, 1)\) where each \(k \in K \setminus \{0\}\) has a multiplicative inverse (i.e., \((K \setminus \{0\}, \odot, 1)\) forms a group). Examples of ordered semifields are the tropical semiring \((\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0, \preceq)\) and the semiring of non-negative rational numbers \((\mathbb{Q}_{\geq 0}, +, \times, 0, 1, \leq)\).

Fix a semiring \((K, \oplus, \odot, 0, 1)\). Let \(X\) be a set of variables and define \(\mathcal{C}(X)\) as the set of captures \(\{x^+, \neg x \mid x \in X\}\). To extend spanners with annotations, we use the formalism of weighted variable set automata (Doleschal et al., 2020) which defines the class of all regular spanners with annotations, also called regular annotators. A weighted variable set automaton (wVA) over \(K\) is a tuple \(\mathcal{A} = (X, Q, \delta, I, F)\) such that \(X\) is a finite set of variables, \(Q\) is a finite set of states, \(\delta : Q \times (\Sigma \cup \mathcal{C}(X)) \times Q \rightarrow K\) is a weighted transition function and \(I : Q \rightarrow K\) and \(F : Q \rightarrow K\) are the initial and final weight functions, respectively. A run \(\rho\) over a document \(d = a_1 \cdots a_n\) is a sequence of the form:

\[
\rho := (q_0, i_0) \xrightarrow{a_1} (q_1, i_1) \xrightarrow{a_2} \cdots \xrightarrow{a_m} (q_m, i_m)
\]

where (1) \(1 = i_0 \leq i_1 \leq \cdots \leq i_m = n + 1\), (2) each \(q_j \in Q\) with \(I(q_0) \neq 0 \neq F(q_m)\), (3) \(\delta(q_j, o_{j+1}, q_{j+1}) \neq 0\), and (4) \(i_{j+1} = i_j\) if \(o_{j+1} \in \mathcal{C}(X)\) and \(i_{j+1} = i_j + 1\) otherwise. In addition, we say that a run \(\rho\) is valid if for every \(x \in X\) there exist exactly one index \(i\) with \(o_i = x^+\), exactly one index \(j\) with \(o_j = \neg x\), and \(i < j\). We denote by \(\text{Run}_\mathcal{A}(d)\) the set of all valid runs of \(\mathcal{A}\) over \(d\). Note that for some wVA \(\mathcal{A}\) and document \(d\) there could exist runs of \(\mathcal{A}\) over \(d\) that are not valid. For this reason, we say that \(\mathcal{A}\) is functional if every run \(\rho\) of \(\mathcal{A}\) over \(d\) is valid for every document \(d\). Given that some decision
problems for non-functional variable-set automata are NP-hard (Freydenberger et al., 2018; Maturana et al., 2018), from now on we assume that all wVAs are functional.

A valid run $\rho$ like above naturally defines a mapping $\nu^\rho$ over $X$ that maps each $x$ to the span $[i_j, i_j']$ where $o_{i_j} = x^+$ and $o_{i_j'} = x^-$. Furthermore, we associate a weight in $K$ to $\rho$ by multiplying all the weights of the transitions as follows:

$$W(\rho) := I(q_0) \odot \bigotimes_{j=1}^m \delta(q_j, o_{j+1}, q_{j+1}) \odot F(q_m).$$

We define the set of output mappings of $\mathcal{A}$ over $d$ as $[\mathcal{A}](d) = \{\nu^\rho \mid \rho \in \text{Run}_{\mathcal{A}}(d)\}$. Given a mapping $\nu \in [\mathcal{A}](d)$, we associate the weight

$$W_{\mathcal{A},d}(\nu) = \bigoplus_{\rho \in \text{Run}_{\mathcal{A}}(d) : \nu = \nu^\rho} W(\rho).$$

Intuitively, each $\nu \in [\mathcal{A}](d)$ contains relevant data extracted by $\mathcal{A}$ from $d$, and $W_{\mathcal{A},d}(\nu)$ is the annotation attached to $\nu$ obtained during the extraction process, e.g. confidence or support.

In (Doleschal et al., 2020), the problem of ranked annotator enumeration was proposed, which for the sake of completeness we present next:\footnote{In (Doleschal et al., 2020) they considered positively ordered semiring, which is slightly more general that the notion of ordered semiring used here.}

<table>
<thead>
<tr>
<th>Problem:</th>
<th>RA-ENUM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>A wV A over an ordered semiring $K$ and a document $d$.</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td>Enumerate all $\nu_1, \ldots, \nu_k \in <a href="d">\mathcal{A}</a>$ without repetitions and such that $W_{\mathcal{A},d}(\nu_1) \preceq W_{\mathcal{A},d}(\nu_{i+1})$.</td>
</tr>
</tbody>
</table>

RA-ENUM was studied in (Doleschal et al., 2020) and an enumeration algorithm was provided with polynomial preprocessing and polynomial delay in terms of $|\mathcal{A}|$ and $|d|$. By using the framework of MSO cost functions, we can give a better algorithm for a special case of RA-ENUM. We say that a wV A is unambiguous if, for every document $d$ and $\nu \in [\mathcal{A}](d)$, there exists at most one run $\rho \in \text{Run}_{\mathcal{A}}(d)$ such that $\nu = \nu^\rho$. The connection between cost transducers and wV A is direct, although the
former works over groups and wVA works over semirings. For this reason, we restrict wVA to semifields and give the following result.

**Corollary 7.** The problem RA-ENUM can be solved with $|A| \cdot |d|$ preprocessing time and $\log(|A| \cdot |d|)$-delay when $A$ is unambiguous and $K$ is an ordered semifield.

Although the previous result is a restricted case of RA-ENUM and a direct consequence of Theorem 10, to the best of our knowledge this is the first non-trivial ranked enumeration algorithm proposed for the framework of document spanner.

### 5.4. Ranked Enumeration Algorithm

In this section, we will see how novel data structures can solve the ranked enumeration problem for cost transducers on words. We provide an algorithm for the RANK-ENUM-T problem, which uses a structure called Heap of Words (HoW) as a black box. We specify the interface of the HoW, to then present the full algorithm. The HoW structure is addressed in detail in the next section. This structure has the property of being fully-persistent. Given that this is a crucial property, we start with a brief introduction to this concept.

**Fully-persistent data structures.** A data structure is said fully-persistent (Driscoll et al., 1989) when no operation can modify the data structure. In a fully-persistent data structure, all the operations return new data structures, without changing the original ones. While this seems to be a restriction on the possible operations, it allows “sharing”. For instance, with a fully-persistent linked list data structure, we can keep two lists $l_1, l_2$ with $l_1$ being some value followed by the content of $l_2$ and since no operation modifies the content of $l_1$ or $l_2$ there is no risk that an access to $l_1$ modifies indirectly $l_2$. In contrast, if we had allowed an operation that modifies the first value of a list in place (i.e., without returning a new list containing the modification), the applying this new operation on $l_2$ would have modified both $l_1$ and $l_2$.

All data structures that we use in this chapter are fully-persistent. We use these data structures to store and enumerate the outputs of the cost transducer while, at the
same time, share and modify the outputs without any risk of losing them. For more information of fully-persistent data structures, we refer the reader to (Driscoll et al., 1989).

The HoW data structure. A Heap of Words (HoW) over an ordered group \((G, \oplus, O, \preceq)\) is a data structure \(h\) that stores a finite set \([w_1 : g_1], \ldots, [w_n : g_n]\) where each \([w_i : g_i]\) is a pair composed by a word \(w_i \in \Sigma^*\) and a priority \(g_i \in G\). Further, we assume that \(w_i \neq w_j\) for every \(i \neq j\), namely, the stored words form a set too. We define \([h] = \{w_1, \ldots, w_n\}\) as the content of \(h\) and, given the previous restriction, there is a one-to-one correspondence between \([w_i : g_i]\) and \(w_i\). Notice that we will usually write \(h = \{[w_1 : g_1], \ldots, [w_n : g_n]\}\) to denote that \(h\) stores \([w_1 : g_1], \ldots, [w_n : g_n]\) but, strictly speaking, \(h\) is a data structure (i.e., a heap). Finally, we denote by \(\emptyset\) the empty HoW.

The purpose of a HoW \(h\) is to store words and retrieve quickly the pair \([w : g]\) with minimum priority with respect to the order \(\preceq\) of the group. We also want to manage \(h\) by deleting the word with minimum priority, adding new words, increasing the priority of all elements by some \(g \in G\), or extending all words with a new letter \(a \in \Sigma\). Furthermore, we want to build the union of two HoWs. More formally, we consider the following set of functions to manage HoWs. For HoWs \(h_1\) and \(h_2\), \(w \in \Sigma^*, g \in G\), and \(a \in \Sigma\) we define:

\[
\begin{align*}
w' & := \text{FINDMIN}(h) & h' & := \text{MELD}(h_1, h_2) \quad \text{s.t. } [h_1] \cap [h_2] = \emptyset \\
h' & := \text{DELETEMIN}(h) & h' & := \text{ADD}(h, [w : g]) \quad \text{s.t. } w \notin [h] \\
h' & := \text{INCREASEBY}(h, g) & h' & := \text{EXTENDBY}(h, a)
\end{align*}
\]

where \(h'\) is a new HoW and \(w' \in \Sigma^*\). In general, each of such functions receives a HoW and outputs a HoW \(h'\). As it was explained before, this data structure is fully-persistent and, therefore, after applying any of these functions, both the output \(h'\) and its previous version \(h\) are available. Now, we define the semantics of each operation.

Let \(h = \{[w_1 : g_1], \ldots, [w_n : g_n]\}\). The \text{FINDMIN} of \(h\) returns a word \(w'\) such that \([w' : g']\) is stored in \(h\) and \(g'\) is minimal among all the priorities stored, formally, \([w' : g'] \in h\) and \(g' = \min\{g \mid [w : g] \in h\}\). If there are several \(w'\) satisfying this property, one is picked arbitrarily. Operation \text{DELETEMIN} returns a new HoW \(h'\) that stores the set represented by \(h\) without the pair of the word returned by \text{FINDMIN}(h), that is,
Algorithm 5 Preprocessing phase for RANK-ENUM-T.

Require: \( T = (Q, \Delta, \kappa, I, F) \) and \( w = a_1 \ldots a_n \).

1: procedure \textsc{Preprocessing}(\( T, w \))
2: \hspace{1em} for all \( q \in \text{dom}(I) \) do
3: \hspace{2em} \( h_q^0 \leftarrow \text{ADD}(\emptyset, [\epsilon : I(q)]) \)
4: \hspace{1em} \text{for } i \text{ from } 1 \text{ to } n \text{ do}
5: \hspace{2em} \text{for all } \( t = (p, a, \bar{X}, q) \in \Delta \) do
6: \hspace{3em} \( h \leftarrow h^{i-1}_p \)
7: \hspace{3em} if \( \bar{X} \neq \emptyset \) then
8: \hspace{4em} \( h \leftarrow \text{EXTENDBY}(h, (\bar{X}, i)) \)
9: \hspace{3em} \( h \leftarrow \text{INCREASEBY}(h, \kappa(t)) \)
10: \hspace{3em} \( h^i_q \leftarrow \text{MELD}(h^i_q, h) \)
11: \hspace{1em} \text{for all } q \in \text{dom}(F) \text{ do}
12: \hspace{2em} \( h \leftarrow \text{INCREASEBY}(h^n_q, F(q)) \)
13: \hspace{2em} \( h_{\text{out}} \leftarrow \text{MELD}(h_{\text{out}}, h) \)
14: \hspace{1em} \text{return } h_{\text{out}}

\[ h' = h \setminus \{[w': g']\} \text{ where } w' = \text{FindMin}(h) \text{ and } g' = \min\{g \mid [w : g] \in h\}. \]

Finally, the functions \textsc{ADD}, \textsc{INCREASEBY}, \textsc{EXTENDBY}, and \textsc{MELD} are formally defined as:

\[
\text{MELD}(h_1, h_2) := h_1 \cup h_2
\]

\[
\text{ADD}(h, [w : g]) := h \cup \{[w : g]\}
\]

\[
\text{INCREASEBY}(h, g) := \{[w_1 : (g_1 \oplus g)], \ldots, [w_n : (g_n \oplus g)]\}
\]

\[
\text{EXTENDBY}(h, a) := \{[(w_1 \cdot a) : g_1], \ldots, [(w_n \cdot a) : g_n]\}
\]

We assume that \textsc{ADD}, \textsc{INCREASEBY}, \textsc{EXTENDBY} and \textsc{MELD} take constant time and \textsc{FindMin} takes \( O(|w'|) \) where \( w' = \text{FindMin}(h) \). For \textsc{DeleteMin}(h), if \( h \) was built using \( n \) operations \textsc{ADD}, \textsc{INCREASEBY}, \textsc{EXTENDBY} and \textsc{MELD} followed by some number of operations \textsc{DeleteMin} then computing \textsc{DeleteMin}(h) takes \( O(|w'| \cdot \log(n)) \) where \( w' = \text{FindMin}(h) \). In the next section we show how to implement \textsc{HoW}s in order to satisfy these requirements. For now, we assume the existence of this data structure and use it to solve RANK-ENUM-T.

The algorithm. In Algorithms 5 and 6, we show the preprocessing and enumeration phases to solve RANK-ENUM-T, respectively. On one hand, the \textsc{Preprocessing}
Algorithm 6 Enumeration phase for RANK-ENUM-T.

Require: A heap of words $h$.

1: procedure ENUMERATION($h$)
2: write 
3: while $h \neq \emptyset$ do
4: write FINDMIN($h$)
5: $h \leftarrow$ DELETEMIN($h$)
6: write 

procedure receives a cost transducer $\mathcal{T} = (Q, \Delta, \kappa, I, F)$ and a word $w \in \Sigma^*$, and computes a HoW $h_{\text{out}}$. On the other hand, the ENUMERATION procedure receives a HoW (i.e., $h_{\text{out}}$) and enumerates $\text{enc}(\sigma_1), \ldots, \text{enc}(\sigma_k)$ such that $\{\sigma_1, \ldots, \sigma_k\} = \exists \mathcal{T}(w)$ and $\text{cost}_\mathcal{T}(w, \sigma_i) \leq \text{cost}_\mathcal{T}(w, \sigma_{i+1})$.

In both procedures we use HoW to compute the set of answers. Indeed, for each $q \in Q$ and each $i \in \{0, \ldots, |w|\}$ we compute a HoW $h_q^i$, and also compute a $h_{\text{out}}$ to store the final results. We assume that all HoWs are empty (i.e., $h_{\text{out}} = \emptyset$ and $h_q^i = \emptyset$) when the algorithm starts. For each $i$, we call the set $\{h_q^i \mid q \in Q\}$ the $i$-level of HoW. Starting from the $0$-level (lines 2-3), the preprocessing phase goes level by level, updating the $i$-level with the previous $(i-1)$-level (lines 4-10). Note here that the $\text{MELD}(h_q^i, h)$ call (line 10) is well-defined since $\mathcal{T}$ is unambiguous (i.e., $[h_q^i] \cap [h] = \emptyset$). After reaching the last $n$-level, the algorithm joins all HoWs $\{h_q^n \mid q \in \text{dom}(F)\}$ into $h_{\text{out}}$, by incrementing first their cost with $F(q)$ and melding them into $h_{\text{out}}$ (lines 11-13). Finally, the preprocessing phase return $h_{\text{out}}$ as output (line 14).

In order to understand the preprocessing algorithm, one has to notice that all the evaluation is based on a very simple fact. Let $w_i = a_1 \ldots a_i$ and define the set $\text{Run}_\mathcal{T}(q, w_i)$ of all partial runs of $\mathcal{T}$ over $w_i$ that end in state $q$. For any of such runs $\rho = q_0 a_1/X_1^i \ldots a_i/X_q^i \in \text{Run}_\mathcal{T}(q, w_i)$, define the partial cost of $\rho$ as $\kappa^*(\rho) = I(q_0) \oplus \bigoplus_{j=1}^i \kappa((q_j-1, a_j, X_j, q_j))$. After executing $\text{PREPROCESSING}$, it will hold that: $h_q^i = \{ \text{out}(\rho) : \kappa^*(\rho) \mid \rho \in \text{Run}_\mathcal{T}(q, w_i) \}$. This is certainly true for $h_q^0$ after lines 2-3 are executed. Then, if this is true for $(i-1)$-level, after the $i$-th iteration of lines 5-10 we will have that $h_q^i$ contains all pairs of the form $[\text{out}(\rho) \cdot (X, i) : \kappa^*(\rho) \oplus \kappa(t)]$ for each $t = (p, a_i, X, q) \in \Delta$, plus all pairs $[\text{out}(\rho) \cdot \kappa^*(\rho) \oplus \kappa(t)]$ for each $t = (p, a_i, \emptyset, q) \in \Delta$.
and $\rho \in \text{Run}_T(p, w_{i-1})$. Given that each line takes constant time, we can conclude that the preprocessing phase takes time $O(|T| \cdot |w|)$ as expected.

For the enumeration phase, we extract each output from $h_{\text{out}}$, one by one, by alternating between the $\text{FindMin}$ and $\text{DeleteMin}$ procedures. Since with $\text{DeleteMin}$ we remove the minimum element of $h$ after printing it, the correctness of the enumeration phase is straightforward. Notice that this enumeration will print all outputs in increasing order of priority. Furthermore, it will not print any output twice given that $h_{\text{out}}$ contains no repetitions. To bound the time, notice that the number of $\text{Add}$, $\text{IncreaseBy}$, $\text{ExtendBy}$ and $\text{Meld}$ functions used during the pre-processing is at most $O(|T| \cdot |w|)$. For this reason, the delay between each output $w'$ is bounded by $O(\log(|T| \cdot |w|) \cdot |w'|)$, satisfying the promised delay between outputs.

We want to finish this section by emphasizing that the ranked enumeration problem of cost transducers reduces to computing efficiently the HoW’s methods. Moreover, it is crucial in this algorithm that this data structure is fully-persistent, and each operation takes constant time. Indeed, this allows us to pass the outputs between levels very efficiently and without losing the outputs of the previous levels.

5.5. The Implementation of HoW Data Structure

In this section we focus on the HoW data structure and explain its implementation using yet another structure called incremental Brodal queue. We begin by explaining the general technique we use to store sets of strings with priorities, to then give a full implementation of the functions to manage HoWs with its corresponding complexity analysis.

Let $\Sigma$ be a possibly infinite alphabet and $\mathbb{G} = (\mathbb{G}, \oplus, O, \preceq)$ an order group. A string-DAG over $\Sigma$ and $\mathbb{G}$ is a DAG $D = (V, E)$ where the edges are annotated with symbols in $\Sigma \cup \{\epsilon\}$ and priorities in $\mathbb{G}$. Formally, each edge has the form $e = (u, a, g, v)$, where $u, v \in V$, $a \in \Sigma \cup \{\epsilon\}$ and $g \in \mathbb{G}$. Given a path $\rho = v_1 \xrightarrow{a_1, g_1} \ldots \xrightarrow{a_k, g_k} v_k$, let $[w_\rho, g_\rho]$ be the pair defined by $\rho$, where $w_\rho = a_1 \ldots a_k$ and $g_\rho = g_1 \oplus \ldots \oplus g_k$. We make two more assumptions that any string-DAG must satisfy. First, we assume that there is a special sink vertex $\bot \in V$ that is reachable from any
$v \in V$, has no outgoing edges, and that all edges with $\epsilon$ must point to $\perp$. Second, we assume that, for every $v \in V$ and every two different paths $\rho$ and $\rho'$ from $v$ to $\perp$, it holds that $w_{\rho} \neq w_{\rho'}$. Given these two assumptions, we say that each $v \in V$ encodes a set of pairs $[D](v)$: for $v = \perp$ this set is the empty set, while for all $v \neq \perp$ this set is defined by all the paths from $v$ to $\perp$, i.e., $[D](v) = \{[w_{\rho} : g_{\rho}] | \rho$ is a path from $v$ to $\perp\}$. By these two assumptions, there is a correspondence between the words in $[D](v)$ and the paths from $v$ to $\perp$. For instance, the strings associated with $n_0$ in the string-DAG depicted in Figure 5.1a are $ad$ with priority $0 + 3 = 3$, $abc$ with priority $0 + 1 + 2 = 3$, $aec$ with priority $0 + 4 + 2 = 6$ and $\epsilon$ with priority 5.

This structure is useful to store a big number of strings in a compressed manner. Further, since $\epsilon$ can only appear at the last edge of a path, by doing a DFS it can be used to retrieve all of them without repetitions and taking time linear in the length of each string. However, one can see that it is not very useful when we want to enumerate them by rank order. This motivates the following string-DAG construction. We define a function $\text{prioritize}(D)$ that receives a string-DAG $D = (V, E)$ and returns a string-DAG $D' = (V, E')$ where each edge $(u, a, g, v)$ of $E$ is replaced by an edge $(u, a, g \oplus g', v)$ in $E'$, where $g'$ is the minimum priority in $[D](v)$. For instance, Figure 5.1b shows the string-DAG resulting after applying $\text{prioritize}$ to $D$ of Figure 5.1a. Having $\text{prioritize}(D)$ makes finding the string with minimum priority of a vertex much easier: we simply need to follow recursively the edge with minimum priority. In $n_0$ of Figure 5.1b we make the path $n_0 \xrightarrow{a, 3} n_1 \xrightarrow{b, 3} n_2 \xrightarrow{c, 2} \perp$ and compute the minimum pair $[abc, 3]$ (the priority is retrieved from the first edge).
Before presenting the HoW implementation, we need to introduce another fully-persistent data structure. This structure is based on the Brodal queue (Brodal & Okasaki, 1996), a known worst-case efficient priority queue, which we extend with the new function \texttt{increaseBy}. Formally, an \textit{incremental Brodal queue}, or just a \textit{queue}, is a fully-persistent data structure \(Q\) which stores a set \(P = \{[E_1 : g_1], \ldots, [E_k : g_k]\}\), where each \(E_i\) is a stored element and \(g_i\) is its priority. As an abuse of notation, we often write \(Q = P\). The functions to manage incremental Brodal queues include all functions for HoW except \texttt{EXTENDBY}, namely \texttt{findMin}, \texttt{deleteMin}, add, \texttt{increaseBy} and \texttt{meld}; their definition also remains the same as for HoW. Note that we use different fonts to distinguish the operations over HoWs versus the operations over incremental Brodal queues. For example, we write \texttt{FINDMIN} for HoWs and \texttt{findMin} for queues. Further, this queue has two additional functions: \texttt{isEmpty}, that checks if the queue is the empty queue \(\emptyset\); and \texttt{minPrio}, that returns the value \(g\) of the minimal priority among all the priorities stored. For the rest of this section we assume the existence of an incremental Brodal queue structure such that all functions run in time \(O(1)\) except for \texttt{deleteMin}, which runs in \(O(\log(n))\), where \(n\) is the number of pairs stored in the queue. Finally, all these operations are fully-persistent. The in-detail explanation of this structure is derived to the next section.

With the previous intuition and the structure above, we can now present the implementation for Heap of Words. A HoW \(h\) is implemented as an incremental Brodal queue \(Q\) that stores a set \(\{(a_1, h_1) : g_1], \ldots, (a_k, h_k) : g_k\}\), where each \(a_i \in \Sigma \cup \{\epsilon\}\), each \(h_i\) is a HoW and each \(g_k \in \mathbb{G}\). We write \(h = \langle Q \rangle\) to make clear that we are talking about a HoW and not the queue. The empty HoW is simply the empty queue \(\langle \emptyset \rangle\). Intuitively, the recursive references to HoWs are used to encode a string-DAG \(D\); more specifically, we use it to encode \(\text{prioritize}(D) = (V, E)\) and store the edges using the queue structure. For every \(u \in V\), we define a HoW \(h_u = \langle Q \rangle\) such that each pair \([(a, h_v) : g]\) stored in \(Q\) represents an edge \((u, a, g, v) \in E\). For instance, continuing with the example of Figure 5.1b, we have a HoW for each vertex: \(h_{\bot} = \langle \emptyset \rangle\), \(h_{n_2} = \langle \{(c, h_{\bot}) : 2\}\rangle\), \(h_{n_1} = \langle \{(e, h_{n_2}) : 6], [(b, h_{n_2}) : 3], [(d, h_{\bot}) : 3]\}\rangle\) and \(h_{n_0} = \langle \{(a, h_{n_1}) : 3], [(e, h_{\bot}) : 5]\}\rangle\).
Algorithm 7 HoW’s implementation of ADD, EXTENDBY, FINDMIN and DELETEMIN.

1: procedure ADD(⟨Q⟩, [a : g])
2: return ⟨add(Q, [(a, ⟨∅⟩) : g])⟩
3: procedure EXTENDBY((Q), a)
4: if isEmpty(Q) then
5: return ⟨∅⟩
6: return ⟨add(∅, [(a, ⟨Q⟩) : minPrio(Q)])⟩
7: procedure FINDMIN((Q))
8: (a, ⟨Q′⟩) ← findMin(Q)
9: if isEmpty(Q′) then
10: return a
11: return FINDMIN(⟨Q′⟩) · a
12: procedure DELETEMIN((Q))
13: if isEmpty(Q) then
14: return ⟨∅⟩
15: (a, ⟨R⟩) ← findMin(Q)
16: Q′ ← deleteMin(Q)
17: ⟨R′⟩ ← DELETEMIN(⟨R⟩)
18: if isEmpty(R′) then
19: return ⟨Q′⟩
20: δ ← minPrio(R′) ⊕ (minPrio(R))⁻¹
21: g ← minPrio(Q) ⊕ δ
22: return ⟨add(Q′, [(a, ⟨R′⟩) : g])⟩

We now explain the implementation of the functions defined in Section 5.4 to manage HoW. Consider a HoW \( h = ⟨Q⟩ \). For each \( \text{OP} \in \{\text{MELD, INCREASEBY}\} \), the function is just applied directly to the queue, i.e., \( \text{OP}(⟨Q⟩) = ⟨\text{op}(Q)⟩ \). The implementation of ADD and the other functions is now described and presented in Algorithm 7.

In the case of ADD(\( h, a \)), an edge is added that points to \( ⟨∅⟩ \); this can be extended to add a word \( w \) instead by allowing that edges keep words instead of single letters. To implement EXTENDBY(\( ⟨Q⟩, a \)), we simply need to create a new queue containing
the element \([(a, \langle Q \rangle) : \text{minPrio}(Q)]\). Note that the appended letter is added not at the end, but at the beginning, meaning that the strings are actually being stored in inverted order. This is managed in \textsc{FindMin}(\langle Q \rangle), where the output string is inverted back. For \textsc{FindMin}(\langle Q \rangle), to get the minimum element we recursively use \text{findMin}(Q) to find the outgoing edge with minimum priority, as we explained when the \text{prioritize} function was introduced. For \textsc{DeleteMin}, in order to delete the string with minimum priority, we use the fact that the set of all paths, minus the one with minimal priority, is composed by: (1) all the paths that do not start with the minimal edge, and (2) all the paths starting with the minimal edge that are followed by any path minus the one with minimal priority. For instance, in Figure 5.1b, the minimal path from \(n_0\) is \(\pi = n_0 \xrightarrow{a} n_1 \xrightarrow{d} \perp\). Then, the set of paths minus \(\pi\) is composed by (1) \(n_0 \xrightarrow{c} \perp\), and (2) \(n_0 \xrightarrow{a} n_1 \xrightarrow{z} n_2 \xrightarrow{c} \perp\). In procedure \textsc{DeleteMin}, \langle Q' \rangle stores the paths of (1), while \langle R' \rangle stores the paths from (2) minus the first edge (lines 16-17). Further, since the minimal path was removed, a new priority needs to be computed for this edge, which is computed and stored as \(g\) (line 20-21). This priority is used to create an edge to \(R'\), i.e., \([(a, \langle R' \rangle) : g]\), which together represent the paths of (2). This is connected with the paths of (1), i.e., \langle Q' \rangle, and the result is returned in line 22. The border case case where (2) is empty is managed by lines 18-19, in which case it simply returns \langle Q' \rangle.

We now check that this data structure achieves the time and space bounds given in Section 5.4. We recall \textsc{Meld} and \textsc{IncreaseBy} are implemented using their incremental Brodal queue equivalent and thus inherit the constant time complexity of those.

As we can see, the functions \textsc{Add} and \textsc{ExtendBy} make a constant number of calls to the constant time functions \text{add}, \text{isEmpty} and \text{minPrio} and thus they take constant time.

The function \textsc{FindMin} is recursive but it will make one recursive call for each letter in the output and at each recursive step it will make one call to \text{findMin} and to \text{isEmpty}, therefore the overall complexity of \textsc{FindMin} is linear in the returned word. Notice that we use the \(\cdot\) operator (to denote concatenation) on strings and we suppose
that it takes constant time. For this, we can encode strings as lists of individual letters in reversed order and thus appending a letter at the end of the word corresponds to appending it at the beginning of a list which can be implemented to take constant time.

Let us now dive into the complexity of \textsc{DeleteMin}. We claimed that the complexity of \textsc{DeleteMin}(h) was $O(|w| \times \log(n))$, where $w = \text{FindMin}(h)$ and $n$ is the number of operations that were used to build $h$ without counting the \textsc{DeleteMin} (these \textsc{DeleteMin} should only happen at the enumeration phase, hence after all the \textsc{Add}, \textsc{IncreaseBy}, \textsc{Meld} and \textsc{ExtendBy}). The complexity of \textsc{DeleteMin} is dominated by the calls to deleteMin, and there are $|w|$ such calls. We thus need to prove that each of these deleteMin takes $O(\log(n))$ time.

The branching factor of a \textsc{HoW} $\langle Q \rangle$, noted \text{branch}(\langle Q \rangle), is recursively defined as the maximum between the number of elements in $Q$ and the branching factors of \textsc{HoW} contained in $Q$. By definition, any deleteMin operation triggered by \textsc{DeleteMin}(h) takes time at most logarithmic in \text{branch}(h), thus it suffices to prove \text{branch}(h) \leq n.

Let $h_0, h_1, \ldots, h_k$ be a sequence of \textsc{HoWs} such that $h_0 = \emptyset$ and each $h_i$ is the result of one of the functions \textsc{Add}, \textsc{IncreaseBy}, \textsc{ExtendBy}, or \textsc{Meld} applied over any of the previous \textsc{HoWs} $h_0, \ldots, h_{i-1}$. We will prove that for all $i$ the queue $h_i$ contains less than $i$ elements. Let us introduce $E_i$ as the set of pairs $\langle (Q), a \rangle$ for which there exists at least one $g$ such that $[(a, \langle Q \rangle) : g]$ belongs to one of the queues $h_0, \ldots, h_{i-1}$. When $h_i$ is built with \textsc{Add} or \textsc{ExtendBy} we have that $|E_{i+1} \setminus E_i| \leq 1$, when $h_i$ is built with \textsc{Meld} or \textsc{IncreaseBy} we have $E_{i+1} = E_i$; all in all, we get $|E_i| \leq i$. Now, clearly the queue $h_i$ contains less than $|E_i|$ elements and it is easy to see that \text{branch}($h_n$) is bounded by $\max_{i \leq n}(|h_i|)$ where $|h_i|$ is the number of elements in the queue of $h_n$ (queues that are not top queues are created by an \textsc{ExtendBy} operation and never modified). Thus, we do have the expected \text{branch}($h_n$) $\leq n$ for an $h_n$ built without using any \textsc{DeleteMin}.

When applying \textsc{DeleteMin}, the branching factor can only decrease because \textsc{DeleteMin}(h) might add an edge to some of the queues in $h$ but only after removing one in those queues. Therefore if $h$ is obtained by using $n$ operations \textsc{Add}, \textsc{IncreaseBy}, \textsc{Meld} and \textsc{ExtendBy} followed by $k$ operations \textsc{DeleteMin}, its
branching factor is bounded by $n$. And that gives us the complexities required in Section 5.4.

We end this section by arguing that the implementation of HoW is fully-persistent. For this, note that the performance of HoW relies on the implementation of incremental Brodal queues. Indeed, given that these queues are fully-persistent and each method in Algorithm 7 creates new queues without modifying the previous ones, the whole data structure is fully-persistent. Therefore, it is left to prove that we can extend Brodal queues as we already mentioned. We will show this in the last section.

5.6. Incremental Brodal Queue

In this section, we discuss how to implement an incremental Brodal queue, the last ingredient of our ranked enumeration algorithms for MSO cost functions. This data structure extends Brodal queues (Brodal & Okasaki, 1996) by including the increaseBy procedure. Indeed, our construction of incremental Brodal queues follows the same approach as in (Brodal & Okasaki, 1996). We start by defining what we call an incremental binomial heap, for which most operations take logarithmic time, to then show how to extend it to lower the cost to constant time, except for deleteMin that takes logarithmic time.

5.6.1. Incremental Binomial Heap

The relevant aspects for this extension (i.e., to support increaseBy) appear in the definition of the incremental binomial heap. For this reason, in this subsection we present only the implementation of the incremental binomial heap. The details of how to extend it to an incremental Brodal queue can be found in the next subsection. We start by introducing some notation.

A multitree structure is a tuple $M = (V, \text{first}, \text{next}, v^0)$ where $V$ is a set of nodes, $\text{first} : V \to V \cup \{\bot\}$ and $\text{next} : V \to V \cup \{\bot\}$ are functions such that $\bot \not\in V$ and $v^0 \in V$ is a special node. Further, we assume that the directed graph $G_M = (V, \{(u, v) \mid \text{first}(u) = v \text{ or } \text{next}(u) = v\})$ is a multitree, namely, it is a directed acyclic graph (DAG) in which the set of vertices reachable from any vertex induces a
tree. Let $V_{v^0}$ denotes the reachable nodes from $v^0$ and $G_{v^0} = (V_{v^0}, \{(u, v) \mid \text{first}(u) = v \text{ or } \text{next}(u) = v\})$ the graph induced by $V_{v^0}$, which is a tree by definition. Note that $G_{v^0}$ is using the first-child next-sibling encoding to form an ordered forest. To see this, let $\text{next}^*(v)$ be the smallest subset of $V$ such that $v \in \text{next}^*(v)$ and $\text{next}(u) \in \text{next}^*(v)$ whenever $u \in \text{next}^*(v)$. Then the set $\text{roots} = \text{next}^*(v^0)$ represents the roots of the forest and for each $v \in V_{v^0}$ the set $\text{children}(v) = \text{next}^*(\text{first}(v))$ are the children of the node $v$ in the forest where $\text{children}(v) = \emptyset$ when $\text{first}(v) = \bot$. Here both sets are ordered by the next function, then we will usually write $\text{roots} = v_1, \ldots, v_j$ or $\text{children}(v) = u_1, \ldots, u_k$ to denote both the elements of the set and its order. Also, we write $\text{parent}(v) = u$ if $v \in \text{children}(u)$ and we say that $v$ is a leaf if $\text{first}(v) = \bot$. Note that in $M$ a node could have different “parents” (i.e., $G_M$ is a DAG) depending on the node $v^0$ that we start. We say that $M$ forms a tree if $\text{next}(v^0) = \bot$. Furthermore, for $v \in V$ we denote by $M_v$ the tree hanging from $v$, namely, $M_v$ is equal to $M$ with the exception that $v_{0_M}^0 = v$ and $\text{next}_M(v) = \bot$. As it will clear below, this encoding will be helpful to build the data structure and assure the persistent requirement.

A binomial tree of rank $k$ is recursively defined as follows. A binomial tree of rank 0 is a leaf and a binomial tree of rank $k + 1$ is a multtree structure $M$ that forms a tree such that $\text{children}(v^0) = u_k, \ldots, u_0$ and $M_u_i$ is a binomial tree of rank $i$. If $M$ is a binomial tree we denote its rank by $\text{rank}(M)$. One can easily show by induction over the rank (see (Cormen, Leiserson, Rivest, & Stein, 2009)) that for every binomial tree $M$ of ranked $k$, it holds that $|V_{v^0}| = 2^k$ and, thus, the number of children of each node is of logarithmic size with respect to the size of $T$, i.e., $|\text{children}(v)| \leq \log(|V_{v^0}|)$ for every $v \in V_{v^0}$. We use this property several times throughout this section.

Fix an ordered group $(\mathbb{G}, \oplus, 0, \preceq)$. An incremental binomial heap over $\mathbb{G}$ is defined as a tuple $H = (V, \text{first}, \text{next}, v^0, \Delta, \text{elem}, \delta^0)$ where $(V, \text{first}, \text{next}, v^0)$ is a multtree structure, $\Delta : V \rightarrow \mathbb{G}$ is the delta-priority function, $\text{elem} : V \rightarrow \mathcal{E}$ is the element function where $\mathcal{E}$ is the set of elements that are stored, and $\delta^0 \in \mathbb{G}$ is an initial delta value. Further, if $M$ is the multtree structure defined by $(V, \text{first}, \text{next}, v^0)$ and $\text{roots} = v_1, \ldots, v_n$ are its roots, then each $M_{v_i}$ is a binomial tree with $\text{rank}(M_{v_i}) < \text{rank}(M_{v_{i+1}})$ for each $i < n$. In other words, an incremental binomial heap has the
same underlying structure than a standard binomial heap (Cormen et al., 2009). Usually in the literature (Brodal & Okasaki, 1996), a binomial heap is imposed a min-heap property, meaning that a node always has lower priority than its children, which is crucial for dequeuing elements in order. Instead, we give to our heap a different semantics by keeping the difference between nodes with the $\Delta$-function and computing the real priority function $\text{pr}_{v,0} : V_{v,0} \to \mathbb{G}$ as follows: $\text{pr}_{v,0}(v) := \delta^0 \oplus \Delta(v)$ whenever $v$ is a root of the underlying multitree structure, and $\text{pr}_{v,0}(v) := \text{pr}_{v,0}(u) \oplus \Delta(v)$ whenever $\text{parent}(v) = u$. Given that $\text{parent}(v)$ depends on the starting node $v^0$, then $\text{pr}_{v,0}$ also depends on $v^0$. In addition, we assume that a min-heap property is satisfied over the real priority function, namely, $\text{pr}_{v,0}(u) \preceq \text{pr}_{v,0}(v)$ whenever $\text{parent}(v) = u$. Then $H$ is a heap where each node $v \in V$ keeps a pair $(\text{elem}(v), \text{pr}_{v,0}(v))$ where $\text{elem}(v)$ is the stored element and $\text{pr}_{v,0}(v)$ its priority in the heap. This principle of storing the deltas between nodes instead of the real priority is crucial for supporting the increased-by operation of the data structure.

Next, we show how to implement the operations of an incremental Brodal queue stated in Section 5.5, namely, $\text{isEmpty}$, $\text{increaseBy}$, $\text{findMin}$ $\text{minPrio}$, $\text{deleteMin}$, $\text{add}$, and $\text{meld}$. We implement this with an incremental binomial heap where the only difference is that $\text{isEmpty}$ and $\text{increaseBy}$ will take constant time, and $\text{findMin}$ $\text{minPrio}$, $\text{deleteMin}$, $\text{add}$, and $\text{meld}$ will take logarithmic time. To extend incremental binomial heaps to lower the complexity of $\text{findMin}$ $\text{minPrio}$, $\text{deleteMin}$, and $\text{add}$ to constant time, one can use the same techniques as in (Brodal & Okasaki, 1996). Most operations of incremental binomial heaps are similar to the operations on binomial heaps (see (Cormen et al., 2009)), however, for the sake of completeness we explain each one in detail, highlighting the main differences to manage the delta priorities.

Let us fix an incremental binomial heap $H = (V, \text{first}, \text{next}, v^0, \Delta, \text{elem}, \delta^0)$. Given that all operations must be persistent, we will usually create a copy $H'$ of $H$ by extending $H$ with new fresh nodes. More precisely, we will say that $H'$ is an extension of $H$ (denoted by $H \subseteq H'$) iff $V_H \subseteq V_{H'}$ and $\text{op}_{H'}(v) = \text{op}_H(v)$ for every $v \in V_H$ and $\text{op} \in \{\text{first}, \text{next}, \Delta, \text{elem}\}$ (note that $v^0$ and $\delta^0$ may change). Furthermore, for $H \subseteq H'$ we will say that a node $v' \in V_{H'} \setminus V_H$ is a fresh copy of $v \in V_H$ if $v'$
in $H'$ has the same structure as $v$ in $H$ where only the differences are defined explicitly, namely, we omit the functions that are the same as for $v$. For example, if we say that “$v'$ is a fresh copy of $v$ such that $\text{next}_{H'}(v') := \bot$”, this means that $\text{next}_{H'}(v') := \bot$ and $\text{op}_{H'}(v') = \text{op}_H(v)$ for every $\text{op} \neq \text{next}$.

The first operation, $\text{isEmpty}(H)$, can easily be implemented in constant time, by just checking whether $v^0 = \bot$ or not. Similarly, $\text{increaseBy}(H, \delta)$ can be implemented in constant time by just updating $\delta^0$ to $\delta^0 \oplus \delta$, which is the purpose of having $\delta^0$. For $\text{findMin}(H)$ or $\text{minPrio}(H)$, a bit more of work is needed. Recall that a $k$-rank binomial tree with $|V|$ nodes satisfies $|V| = 2^k$. Given that $\text{roots} = v_1, \ldots, v_n$ is a sequence of binomial trees ordered by rank, one can easily see that $n \in \mathcal{O}(\log(|V_{\text{root}}|))$. Therefore, we need at most a logarithmic number of steps to find the node $v_i$ with the minimum priority and return $\text{elem}(v_i)$ or $\text{pr}_{v_i}(v_i)$ whenever $\text{findMin}(H)$ or $\text{minPrio}(H)$ is asked, respectively.

For $\text{add}(H, e, g)$ or $\text{deleteMin}(H)$, we reduce them to melding two heaps. For the first operation, we create a heap $H'$ whose multitree structure has one node, call it $v$, $\Delta_{H'}(v) := g$, $\text{elem}_{H'}(v) := e$, and $\delta_{H'}^0 := 0$. Then we apply $\text{meld}(H, H')$ obtaining a heap where the new node $(e, g)$ is added to $H$. For the second operation, we remove the minimum element by creating two heaps and then apply the meld operation. Specifically, let $\text{roots} = v_1, \ldots, v_n$ be the roots of $H$ and $v_i$ be the root with the minimum priority. Then we build two heaps $H_1$ and $H_2$ such that $H \subseteq H_i$ for $i \in \{1, 2\}$. For $H_1$, we extend $H$ by creating fresh copies of all $v_j$, $j \neq i$. Formally, define $V_{H_1} = V_H \cup \{v'_1, \ldots, v'_n\}$ where each $v'_j$ is a fresh copy of $v_j$ with the exception of $v'_{i-1}$ that we set $\text{next}_{H_1}(v'_{i-1}) := v'_{i+1}$. Finally, define $v^0_{H_1} = v'_1$ as the starting node of $H_1$. Now, for $H_2$ we extend $H$ by creating a copy of the children of $v_i$ in $H$ in reverse order and updating $\delta_{H}^0$ to $\delta_{H}^0 \oplus \Delta_H(v_i)$ (recall that the children of a binomial tree are ordered by decreasing rank). Formally, if $\text{children}_H(v_i) = u_1, \ldots, u_k$, then $V_{H_2} = V_H \cup \{u'_1, \ldots, u'_k\}$ where each $u'_j$ is a fresh copy of $u_j$ such that $\text{next}_{H_2}(u'_j) := v'_{j-1}$ for $j > 1$ and $\text{next}_{H_2}(u'_1) := \bot$. Finally, define $v^0_{H_2} := v'_k$ and $\delta_{H_2}^0 := \delta_{H}^0 \oplus \Delta_H(v_i)$. The reader can check that $H_1$ and $H_2$ are valid incremental binomial heaps and, furthermore, $H_1$ is $H$ without $v_i$ and $H_2$ contains only the children of $v_i$ in reverse
order. Therefore, to compute \text{deleteMin}(H) we return \text{meld}(H_1, H_2). Given that the construction of \(H_1\) and \(H_2\) takes at most logarithmic time in the size of \(H\) (i.e., there is at most a log number of roots or children), then the procedure takes logarithmic time. Furthermore, \(H\) was never touched and then the operation is fully-persistent.

For \text{meld}(H_1, H_2), we use the same algorithm as for melding two binomial heaps with two modifications that are presented here. For melding two binomial heaps, we point the reader to (Cormen et al., 2009) in which this operation is well explained. For the first change, we need to update the link operation (Cormen et al., 2009) of two binomial trees to support the use of the delta priorities. Given a incremental binomial heap \(H\) and its underlying multitree structure \(M\), let \(v_1\) and \(v_2\) be two nodes in \(H\) such that \(\Delta(v_1) \leq \Delta(v_2)\) and \(M_{v_1}\) and \(M_{v_2}\) has the same rank \(k\). Then the link of \(v_1\) and \(v_2\), denoted by \(\text{link}(H, v_1, v_2)\), outputs a pair \((H', v'_1)\) such that \(H'\) is an extension of \(H\) and \(M'_{v_1}\) is a binomial tree of rank \(k+1\) containing the nodes of \(M_{v_1}\) and \(M_{v_2}\). Formally, \(V_{H'} := V_H \cup \{v'_1, v'_2\}\) and \(v'_1\) and \(v'_2\) are fresh copies of \(v_1\) and \(v_2\) such that \(\text{first}_{H'}(v'_1) := v'_2, \text{next}_{H'}(v'_2) := \text{first}_H(v_1)\) and \(\Delta_{H'}(v'_2) := \Delta_H(v_1)^{-1} \oplus \Delta_H(v_2)\). Note that the new node \(v'_1\) defines a binomial tree \(M'_{v'_1}\) of rank \(k + 1\) containing all nodes of \(M_{v_1}\) and \(M_{v_2}\), maintaining the priorities of \(H\) and such that \(\text{pr}_{H'}(u) \leq \text{pr}_{H'}(u')\) whenever \(u = \text{parent}(u')\). The second change of the algorithm in (Cormen et al., 2009) is that, before melding \(H_1\) and \(H_2\), we push each initial delta value to the roots of the corresponding data structures. For this, given an incremental binomial heap \(H\) we construct \(H^k\) with \(H \subseteq H^k\) as follows. Let \(\text{roots}_H = v_1, \ldots, v_k\). Then \(V_{H^k} = V_H \cup \{v'_1, \ldots, v'_k\}\) where \(v'_1, \ldots, v'_k\) are fresh copies of \(v_1, \ldots, v_k\) and \(\Delta_{H^k}(v'_i) := \delta^0_{H} \oplus \Delta_H(v_i)\). Furthermore, we define \(v^0_{H^k} := v'_1\) and \(\delta^0_{H^k} := 0\). Note that in \(H^k\) we can forget about the initial delta value given that this is included in the root of each binomial tree. Finally, to meld \(H_1\) and \(H_2\) we construct \(H^k_1\) and \(H^k_2\) and then apply the melding algorithm of (Cormen et al., 2009) with the updated version of the link function, \(\text{link}(H, v_1, v_2)\). Overall, the operation takes logarithmic time to build \(H^k_1\) and \(H^k_2\), and logarithmic time to meld both heaps. Moreover, given that \(\text{link}(H, v_1, v_2)\) and the construction of \(H^k_1\) and \(H^k_2\) do not modify the initial heap \(H\), then the meld operation is persistent as well.
5.6.2. Incremental Brodal Queue

Here we present the complete incremental Brodal queue. In the following, we use a similar approach as the one used in (Brodal & Okasaki, 1996): we start with the incremental binomial heap, which supports most operations in \(O(\log n)\); then, we explain each modification made to this structure until we have built the final structure.

5.6.2.1. Skew incremental binomial heap

We now explain how the time of add can be reduced to \(O(1)\), while maintaining the asymptotic time of the other operations. The technique is borrowed from (Brodal & Okasaki, 1996) and modified to handle increaseBy efficiently.

The motivation comes from the skew binary numbers (Myers, 1983), in particular the canonical skew binary numbers, a variation of binary numbers in which all digits are 0 or 1 with the possible exception of the lowest order non-zero digit, which might be 2. A skew binary number \(\alpha = \alpha_n\alpha_{n-1}\ldots\alpha_1\) denotes the integer value \(\sum_{i=1}^{n} a_i(2^i-1)\). This number representation avoids the carry cascading when adding 1 to a number. For example, the number 43 is represented by the skew binary number 10112, and adding 1 to it results in 10120, which is done with a single carry operation.

A skew binomial tree, or skew tree, is a tree with the following definition:

- a skew tree of rank 0 is a leaf.
- a skew tree of rank \(r + 1\) is formed in one of three ways:
  - a simple link, making a skew tree of rank \(r\) the leftmost child of another skew tree of rank \(r\);
  - a type A link, making two skew trees of rank \(r\) the children of a skew tree of rank 0; or
  - a type B link, making a skew tree of rank 0 and a skew tree of rank \(r\) the leftmost children of another skew tree of rank \(r\).

Note that, unlike binomial trees, now a skew binomial tree of rank \(r\) has a less rigid structure and, in particular, the number of contained nodes \(n\) is not fixed. However, it is not hard to see that it is bounded by \(2^r \leq n \leq 2^{r+1} - 1\). Moreover, given a tree \(T\)
with root \( v_7 \) and \( \text{rank}(T) = r \), the number of children of \( v_7 \) is proportional to \( r \), and therefore \( \text{children}(v_7) \) is \( O(\log |V_{v_7}|) \).

A skew incremental binomial heap (skew heap for short) is defined as a tuple \( H = (V, \text{first}, \text{next}, v^0, \Delta, \text{elem}, \delta^0) \), where all components are the same as for incremental binomial heaps, except that if \( \text{roots} = v_1, \ldots, v_n \) are its roots, then each \( M_{v_i} \) is an incremental skew binomial tree with \( \text{rank}(M_{v_i}) < \text{rank}(M_{v_{i+1}}) \) for each \( 2 < i < n \), and \( \text{rank}(M_{v_1}) \leq \text{rank}(M_{v_2}) \). Namely, we allow the two smallest trees to have the same rank.

Consider a skew heap \( H = (V, \text{first}, \text{next}, v^0, \Delta, \text{elem}, \delta^0) \) with \( \text{roots} = v_1, \ldots, v_n \). Functions isEmpty, increaseBy, findMin, minPrio and meld remain the same as for regular heaps.

For \( \text{add}(H, e, g) \), we create a new skew heap \( H' \) with \( H \subseteq H' \) as follows. Let \( v \) be a fresh node with \( \Delta_{H'}(v) := g \oplus \Delta^{-1} \) and \( \text{elem}_{H'}(v) := e \). If all trees \( M_{v_i} \) have different rank, then we add \( v \) at the beginning of the roots, namely \( v \) is made the root of \( H' \) and \( \text{next}_{H'}(v) = v^0 \). If there are two trees \( M_{v_1}, M_{v_2} \) with the same rank, we link \( v \) with \( v_1 + v_2 \) to form a tree of rank \( r + 1 \) as follows. If \( \Delta(v) < \Delta(v_1) \) and \( \Delta(v) < \Delta(v_2) \), then, we make copies \( v_1', v_2' \) of \( v_1 \) and \( v_2 \) with their priorities decreased by \( \Delta_{H'}(v) \), i.e., \( \Delta_{H'}(v_1) = \Delta_H(v_1) \oplus \Delta_{H'}(v)^{-1} \), make them the children of \( v \) and make \( v \) the root of \( H' \) with \( \text{next}_{H'}(v) = \text{next}_{H}(v_2) \). Otherwise, w.l.o.g. consider that \( \Delta(v_1) < \Delta(v_2) \). Then, we make copies \( v_1', v_2' \) of \( v_1 \) and \( v_2 \) and make \( v_2' \) and \( v \) the leftmost children of \( v_1' \), but decreasing their priorities by \( \Delta(v_1) \) in the same way. Note that the first case represents an A link and the latter a B link, and that the new tree is a valid skew tree of rank \( r + 1 \) that stores the same elements as \( v_1 \) and \( v_2 \), plus the new element \( v \). Moreover, since we allow two repeated ranks and we know that there is at most one other tree of rank \( r + 1 \), adding it does not require a chain of links and thus takes time \( O(1) \).

The case of deleteMin follows the same principle: we make a skew heap \( H_1 \) to store copies of the roots except for the minimal one, call it \( v_i \), make a skew heap \( H_2 \) to store the children of \( v_i \), and build the result with \( \text{meld}(H_1, H_2) \). However, building \( H_2 \) requires a little more work. Since \( M(v_i) \) is a skew heap, among their children could
appear up to \( \log n \) skew heaps of rank 0. Then, we build \( H_2 \) by taking the children with rank higher than 0, and then add the remaining ones using add, taking time \( O(\log n) \). Finally, we return \( \text{meld}(H_1, H_2) \), taking overall time \( O(\log n) \).

5.6.2.2. Incremental Brodal queue

Clearly, this structure alone is not enough for what we need, since most operations still take time \( O(\log n) \). To address this, we base one more time on the techniques of (Brodal & Okasaki, 1996), this time on one called “bootstrapping”.

An incremental Brodal queue (queue for short) is either the empty queue \( Q_0 \) or a pair \( Q = ((e, p), H) \), where \( e \) is a stored element, \( p \) is its priority and \( H \) is a skew heap that stores incremental Brodal queues. Given a queue \( Q \), we denote its components with the subscript \( Q \), e.g., \( e_Q, p_Q, H_Q \). Consider some \((Q, p)\) stored in \( H \). Note that, whenever \( H \) is applied an increaseBy operation, \( p \) is increased but the priorities inside \( Q \) are not; instead, we increase them only when \((Q, p)\) is retrieved from \( H \). We use this setting because the increaseBy operation is able to increase priorities stored in \( H \) efficiently, but increasing all priorities inside \( Q \) might result in an unbounded recursion. Priority \( p \) is then going to store the real updated value of \( p_Q \) considering how much the priorities of \( Q \) have been increased by calling increaseBy over \( H \). Note that \( p \) can be seen as having the form \( p_Q \oplus d \) for some \( d \in \mathbb{G} \) that represents such increase and, thus, \( d = p_Q^{-1} \oplus p \). Consequently, each priority in \( Q \) actually represents a priority \( p \oplus d \). The set of elements stored in a skew heap \( H \) and a queue \( Q \) are defined accordingly by the following two-level recursive definition:

\[
[H] = \bigcup_{(Q,p) \in [H]} \{[Q] \oplus (p_Q^{-1} \oplus p)\}
\]

\[
[Q] = \{(e_Q, p_Q)\} \cup [H]
\]

where \([Q] \oplus d = \{(e, p \oplus d) \mid (e, p) \in [Q]\}\) represents that all priorities of \([Q]\) are increased by \( d \). This recursive definition just says that the \([Q]\) consists of all elements stored in \( Q \), including the ones at other queues inside \( Q \). Moreover, \( Q \) is kept so that \((e_Q, p_Q)\) is always the element with minimum priority of \([Q]\).
Consider a queue $Q = ((e_Q, p_Q), H_Q)$, and let $P = ((e_P, p_P), H_P)$ be the resulting queue after applying each operation over $Q$. Then, $Q$ implements the same operations as a heap:

- **isEmpty($Q$)**. It only checks if $Q$ is equal to the empty queue $Q_\emptyset$ and returns accordingly.

- **add($Q$, $[e' : p']$)**. First, if $Q$ is empty, we initialize a new queue $P$ with $(e_P, p_P) = (e', p')$ and $H_P = H_\emptyset$, where $H_\emptyset$ is the empty heap. Otherwise, if $p' < p_Q$, then $e'$ is the new minimum, so we set $(e_P, p_P) = (e', p')$, otherwise we set it to $(e_Q, p_Q)$. In either case, we set $H_P = H_Q$. W.l.o.g., let $(e', p')$ be the one that is not the minimum. In order to add $(e', p')$ to $H_P$, we create a new queue $Q' = ((e', p'), H_\emptyset)$, and then add $Q'$ to $H_P$ with $add(H_P, [Q' : p'])$.

- **findMin($Q$)** and **minPrio($Q$)**. We just return $e_Q$ or $p_Q$, respectively, and keep $P = Q$.

- **deleteMin($Q$)**. It is easy to find $(e_Q, p_Q)$, but to delete it we need to replace it with the next minimum from $H_Q$ afterwards. We find it by running $R = \text{findMin}(H_Q)$, $u = \text{minPrio}(H_Q)$, and then delete it with $H_P = \text{deleteMin}(H_Q)$. Consider that the queue $R$ has the form $((e_R, p_R), H_R)$ and recall that $u$ stores the updated value of $p_R$. Then, the minimum element is $(e_R, u)$, so we set that as our new $(e_P, p_P)$. Notice that we removed queue $R$, meaning that we not only removed the minimum element, but also all elements stored in $H_R$. In order to add them again, we first increase the priorities in $H_R$ by $(p_R^{-1} \oplus u)$ and then we meld it with $H_P$, by doing $H_P = \text{meld}(\text{increaseBy}(H_R, p_R^{-1} \oplus u), H_P)$.

- **meld($Q$, $R$)**. Let queue $R$ have the form $((e_R, p_R), H_R)$. First, we select between $(e_Q, p_Q)$ and $(e_R, p_R)$ the one with the lowest priority; assume w.l.o.g. that it is $(e_Q, p_Q)$. Then, we set $(e_P, p_P) = (e_Q, p_Q)$ and $H_P = \text{add}(H_Q, [R : p_R])$.

- **increaseBy($Q$, $g$)**. We set $H_P = (e_Q, p_P = p_Q \oplus g)$ and $H_P = \text{increaseBy}(H_Q, g)$, that is, just increase $p_Q$ by $g$ and apply $\text{increaseBy}(H, g)$. Recall that the latter increases only the priorities stored directly in $H_Q$, namely each $(Q', p')$.
in $H_Q$ turns into $(Q', p' \oplus g)$ in $H_P$. The priorities of elements inside $Q'$ remain unchanged, and are updated later when $Q'$ is retrieved from $H_P$. Again, the structure is fully persistent because none of the operations modify $Q$. It is not hard to see that now operations add, findMin, meld and increaseBy take time $O(1)$, and deleteMin takes time $O(\log n)$. 
Chapter 6. CONCLUSIONS

In this work we proposed a framework to manage the evaluation of CER that includes both general and specific features. First, we presented the core of the framework: the CEL query language, the CEA automata model, and evaluation algorithms for this model with strong efficiency guarantees, namely, output-linear delay enumeration of the results. All these components have their own arguments for their design, and overall they present a sound solution for the problem of evaluating the CER core.

Afterwards, we presented two further extensions of these framework that support two crucial components of CER: correlation and ranked enumeration of results. To support correlation, we formally defined the CER operator partition-by into CEL, defined the chain-CEA automata model in order to support it, and proposed a new evaluation algorithm which relies on a novel index data structure to manage equalities and disequalities. Overall, we were able to support the correlation expressible by the partition by operator, while maintaining the same efficiency guarantees.

Finally, to provide ranked enumeration of the results, we presented a result on the more general framework that is MSO. We proposed an algorithm to enumerate the answers of MSO queries over words, where the enumeration order is defined by a cost function, that has linear preprocessing and logarithmic delay in the size of the words. We first introduced the notion of MSO cost functions, to then present a ranked enumeration scheme. This scheme relies on a particular data structure called HoW. To implement them, we extended a persistent data structure called Brodal Queue. Thanks to these data structures, we obtained the bounds of our algorithm. We showed how this result can be used to evaluate the CER windowing operator within.

There are several possible ways in which this research can be extended. Regarding correlation, the first one is to find a unified class of queries that includes chain-CEA and hierarchical queries. Indeed there are simple hierarchical queries (e.g., $R(x) \land S(y) \land T(x)$) that are not definably by chain-CEA. Another relevant question is whether partition-by queries with projection can be evaluated efficiently. Chain-CEA forbid the use of projection and it is not clear how to extend the evaluation algorithm to support it.
Projection was a crucial requirement to define selection strategy constructions within the automata model, so this question is closely related with the question of how to extend the algorithm to support selection strategies. Finally, this work studies the streaming evaluation of equality and disequality predicates in CER, but leaves open the evaluation of other predicates, like inequalities.

Regarding ranked enumeration, one would like to find a lower bound that justifies the logarithmic delay, or if it is possible to achieve a better delay. Also, by plugging in another data structure with the same interface as HoW, but with different complexity on its implementation, we believe a trade-off between preprocessing and delay enumeration complexity might be obtained, allowing to move the logarithmic factor from the delay into the preprocessing parametrically. Also, the introduced data structures are general enough to be used as a black box in other areas of research related to enumeration, in order to obtain direct complexity results. In terms of CER, we also leave open the question of how can we express general ranked enumeration queries in CEL, namely, how to extend the logic with a syntax that support user-defined cost functions.

Now discussing the overall CER framework, there are several branches in which this can be extended. Many features were left outside in order to keep the language and analysis simple, including aggregation, consumption policies, degrees of uncertainty, among others (Cugola & Margara, 2012b). The natural next step is to extend CEL gradually with these features to establish a more complete framework for CER. Another relevant research direction is to make a thorough expressiveness analysis to compare the expressive power of CEL with other existing frameworks both from theory and practice.

A question more closely related to the present work is how to build a unified framework that supports both correlation and ranked enumeration simultaneously. This is an interesting question on itself, since the techniques used to support each component are very different and it is not clear how to merge them into a single solution. Finally, translating these results into a prototype could mean a significant next step for the framework. On the one hand, it would provide a baseline over which people from the
CER community can build and, on the other hand, it would allow this solution to be compared against other CER frameworks in practice.
References


tions of regular expressions. In *Proceedings of pods* (pp. 137–149).


