A Formal Framework for Complex Event Recognition

ALEJANDRO GREZ and CRISTIAN RIVEROS, Pontificia Universidad Católica de Chile & Millennium Institute for Foundational Research on Data, Chile
MARTÍN UGARTE, Millennium Institute for Foundational Research on Data, Chile
STIJN VANSUMMEREN, UHasselt – Hasselt University, Belgium

Complex Event Recognition (CER) has emerged as the unifying field for technologies that require processing and correlating distributed data sources in real-time. CER finds applications in diverse domains, which has resulted in a large number of proposals for expressing and processing complex events. Existing CER languages lack a clear semantics, however, which makes them hard to understand and generalize. Moreover, there are no general techniques for evaluating CER query languages with clear performance guarantees.

In this paper, we embark on the task of giving a rigorous and efficient framework to CER. We propose a formal language for specifying complex events, called CEL, that contains the main features used in the literature and has a denotational and compositional semantics. We also formalize the so-called selection strategies, which had only been presented as by-design extensions to existing frameworks. We give insight into the language design trade-offs regarding the strict sequencing operators of CEL and selection strategies.

With a well-defined semantics at hand, we discuss how to efficiently process complex events by evaluating CEL formulas with unary filters. We start by introducing a formal computational model for CER, called complex event automata (CEA), and study how to compile CEL formulas with unary filters into CEA. Furthermore, we provide efficient algorithms for evaluating CEA over event streams using constant time per event followed by output-linear delay enumeration of the results.

CCS Concepts: • **Information systems → Data streams**; • **Theory of computation → Data structures and algorithms for data management**; *Database query languages (principles)*; Automata extensions.

Additional Key Words and Phrases: Complex event recognition, complex event processing, streaming evaluation, constant delay enumeration.

ACM Reference Format:
Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2021. A Formal Framework for Complex Event Recognition. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2021), 49 pages. https://doi.org/10.1145/3485463

1 INTRODUCTION

Complex Event Recognition (CER for short) refers to the activity of identifying, in streams of continuously arriving event data, collections of events that collectively satisfy some pattern. To that purpose, CER systems allow expressing patterns that match incoming events not only on the basis of their content, but also on where they occur in the input stream, and how this order relates to other events in the stream, in addition to other (spatio-temporal) constraints between events.

Authors' addresses: Alejandro Grez, ajgrez@uc.cl; Cristian Riveros, cristian.riveros@uc.cl, Pontificia Universidad Católica de Chile & Millennium Institute for Foundational Research on Data, Santiago, Chile; Martín Ugarte, martin@martinugarte.com, Millennium Institute for Foundational Research on Data, Santiago, Chile; Stijn Vansummeren, stijn.vansummeren@uhasselt.be, UHasselt – Hasselt University, Data Science Institute, Agoralaan, 3540, Diepenbeek, Belgium.

0362-5915/2021/1-ART1 $15.00
https://doi.org/10.1145/3485463

CER systems hence aim to detect situations of interest, thereby giving timely insights for implementing reactive responses to complex events when necessary. CER has been successfully applied in scenarios like maritime monitoring [56], network intrusion detection [54], industrial control systems [43] and real-time Analytics [57]. Prominent examples of CER systems from academia and industry include SASE [63], EsperTech [2], Cayuga [34], and TESLA/T-Rex [30, 31], among others (see [10, 32, 39] for a survey).

The main research focus in these systems has been on practical issues like scalability, fault tolerance, and distribution, with the objective of making CER systems 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 [64]). This is in contrast with the situation in relational database systems research, where there is an established and formally-defined core language for expressing queries in a declarative manner (namely: relational algebra), as well as an accompanying evaluation model (based on dataflow computation) and efficient algorithms for optimizing and executing programs in the evaluation model. Exactly because the relational algebra and dataflow computation are established models to base practical relational database systems on, is it possible to compare and contrast systems and evaluation strategies.

The situation in CER is less well-established. Table 1 lists a set of CER operators that, according to recent surveys in the field [10, 39], can be considered "basic operators that should be present in every CER language" [39] (we will discuss these operators by example in Section 3). Unfortunately, as has been observed several times [16, 30, 38, 65], the existing CER literature lacks a simple, compositional, and denotational semantics for this core language. Indeed, the semantics of these operators is often defined indirectly, by means of examples [5, 29, 51], or by translation into evaluation models [55, 58, 63]. Even when a formal semantics is given (e.g., [9, 13, 19, 30, 34]), this semantics is somehow unsatisfactory because it has unintuitive behavior (e.g., sequencing is non-associative), or is severely restricted (e.g., operators cannot be nested). One symptom of this problem is that iteration, which is fundamental in CER, has not yet been defined successfully as a compositional operator. Since iteration is difficult to define and evaluate, it is usually restricted by not allowing nesting [34, 63]. As a result of these problems, CER languages (and systems) are generally cumbersome to understand and compare.

Towards resolving this situation, as our first contribution we introduce Complex Event Logic (CEL), a formal logic with well-defined compositional and denotational semantics built from the common basic features identified in Table 1.

The first three operators in Table 1, namely sequencing, disjunction, and iteration, are similar to the operators of standard regular expressions: concatenation, union, and Kleene star, respectively. This is not a coincidence since regular expressions (and automata) have frequently been used as inspiration for CER languages, especially the early ones. A notable distinction between the operators of standard regular expressions and those of CER, however, is that the former are meant to recognize strings, whereas the latter are meant to select subsets of events from a stream of events. In other words, regular expressions define a language (or equivalently, have boolean output), whereas CER operators define mappings from streams to sets of complex events (and thus have more complex output). The distinction is important, and has implied in particular that in many CER languages sequencing and iteration are interpreted to be *non-contiguous*. For example, a sequence such as $R; S$ then matches all complex events comprising of an R-event that is followed by an S-event in the stream, where any other event is allowed to occur in between R and S. By contrast, the corresponding regular expression only matches the string RS: nothing is allowed between R and S since concatenation is interpreted to be contiguous. While it is clear that the non-contiguous case can be expressed by means of the contiguous semantics if one has the ability to declare wildcards

Operator	Description	CEL	Unary CEL	CEA	Note
Sequencing	Two patterns following each other temporally	✓	✓	✓	
Disjunction	Either of two patterns occurring	✓	✓	✓	
Iteration	A pattern occurring repeatedly	✓	✓	✓	
Local filters	Filter complex events based on local properties of the constituent events	✓	✓	✓	
Correlation	Filter complex events based on relations among the constituent events	✓	✗	✗	
Conjunction	Matching multiple patterns at the same time, regardless of the temporal relation	✓	✓	✓	
Negation	Absense of a pattern	✓	✓	✓	
Projection	Transforming attribute values of simple events	✗	✗	✗	*a*
Windowing	Limit complex events to a specified time window	✓	✗	✗	*b*
Selection strategies	Control if and how "irrelevant" events may occur	✓	✓	✓	

Table 1. Basic CER operators, and how they are included in the languages considered in this paper. (a) CEL 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 CEL does not have a dedicated windowing operator, windowing is expressible by means of filtering (although non-unary filtering is required). See Section 4 for a formal definition of CEL and in-depth discussion of its features.

($R; .*; S$ interpreted contiguously matches the semantics of $R; S$ interpreted non-contiguously), CER languages in general interpret sequencing (and iteration) non-contiguously by default and rely on so-called *selection strategies* [19, 34, 64] to specify conditions about whether it is allowed to have "irrelevant" events in-between those that explicitly occur in the CER pattern.

As our second contribution, 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, however, is that 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 [1, 6, 34, 55, 58, 64] or in combination with dataflow-like operators [63] or other search strategies [9, 30]. Unfortunately, however, these automata models are complicated [55, 58], informally defined [9, 34] or non-standard [6, 30, 64]. 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.

Given this scenario, we focus on the "regular" fragment of CEL, called *unary CEL*, which is obtained by restricting all filters to be unary, therefore forbidding correlation. As our third contribution, 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 [61] and finite state transducers [20], 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 paper 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 our main contribution, 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 particluar, it has been repeatedly observed (e.g., [39, 64]) that in 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. Our evaluation algorithm, in contrast, is hence, at least theoretically, not prone to this behavior.

All together, this work hence provides a formal, uniform framework for CER, including a CER query language, and, for its regular fragment, a corresponding evaluation model, compilation techniques, and efficient evaluation algorithm.

Organization of the paper. We discuss related work in Section 2. We give an intuitive introduction to CER and CEL in Section 3. In Section 4 we formally present Complex Event Logic (CEL) and its unary fragment. We introduce Complex Event Automata (CEA) in Section 5, and study the compilation of unary CEL formulas into CEA (and vice-versa) in Section 6. We study selection strategies and their compilation into CEA in Section 7. In Section 8 we give insight into the relationship between non-contiguous sequencing and iteration, contiguous sequencing and iteration, and selection strategies. We present the evaluation algorithm for CEA in Section 9. We conclude and discuss future work in Section 10.

2 RELATED WORK

Active Database Management Systems (ADSMS) and Data Stream Management Systems (DSMS) process data streams, and thus are usually associated with CER systems. Both technologies aim to execute relational queries over dynamic data [3, 14, 27]. In contrast, CER systems see data streams as a sequence of events where the arrival order is the guide for finding patterns inside streams (see [32] for a comparison between ADSMS, DSMS, and CER). Since DSMS query languages (e.g., CQL [15]) do not focus on basic CER operators like sequencing and iteration, they are incomparable with our framework.

CER systems are usually divided into three approaches [10, 16, 32, 39]: automata-based, tree-based, and logic-based, with some systems (e.g., [2, 30]) being hybrids. We next situate our work

among the three approaches, and refer the reader to the surveys [10, 16, 32, 39] for in-depth discussion of each class of systems.

Automata-based systems are close to what we propose in this paper. They typically propose a CER query language that is inspired by regular expressions (cf. Table 1), which is evaluated by means of custom automata models. Previous proposals (e.g., SASE [6], NextCEP [58], DistCED [55]) 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 [6] or its semantics is confusing [58]. Other proposals (e.g., CEDR [19], TESLA [30], PBCED [9]) have formal semantics, but they do not include iteration. An exception is Cayuga [33], 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. Moreover, we present the first evaluation of regular CER queries (without correlation and aggregation) that guarantees, under certain assumptions, constant time per event and output-linear delay enumeration of the output.

Tree-based systems [2, 50, 53] typically consider a CER query language that is inspired by regular expressions (like automata-based systems, cf. Table 1) but evaluate queries by constructing and evaluating a tree of CER operators, much like relational database systems evaluate relational algebra queries. Cost models are typically used to identify efficient trees. Again, the semantics of queries is not formally defined declaratively. Moreover, in contrast to the automaton-based evaluation algorithm that we propose in this paper, the evaluation trees do not have formal performance guarantee of constant time per event and output-linear delay enumeration of the output, even for regular CER queries.

Logic-based systems [13, 17, 28] express CER queries as rules in some form of logic, typically rooted in temporal logic or event calculus, and consequently treat CER evaluation as logical inference. Logic-based systems consequently have a formal, declarative semantics; see [18] for a survey. However, iteration is often expressed by means of recursive rules instead of as an individual operator. While such rules can *detect* that a CER pattern applies repeatedly, they typically do not *collect* the participating events as part of the output complex event. The semantics of iteration in logic-based systems is therefore often distinct from ours.

Regarding the expressiveness of CER query languages, the majority of the CER literature focuses on the system aspects of CER rather than on the foundational aspects. Particularly, apart from the work SASE^{++} [64], which considers the descriptive complexity of a core CER language, there is no formal study of the expressiveness of CER languages. It is unfortunate, however, that SASE^{++} [64] lacks a formal definition of the language under study and, in particular, ignores the issues related to the scoping of variables under Kleene closure, as well as the data output capabilities.

This paper unifies and consolidates the CER framework first proposed and studied in [41] and [42]. Specifically, the work in [41] introduced a first version of the framework, while [42] studied the expressive power of two variants of CEL. We unify the notation and formalization of both works basing ourselves on the second version of CEL, called Set CEL in [42], and study its properties with respect to the CEA model proposed in [41]. Furthermore, the comparison between the expressiveness of CEL and CEA was studied in [42]. The present paper unifies and consolidates the framework of both former papers. In addition, we formulate stronger versions of the results first mentioned in [41] and [42], and provide full proofs, which were omitted in those papers. In particular, here we provide a novel version of the evaluation algorithm (Section 9) that has a more direct and formal presentation.

Our complex event automata (CEA) build upon the rich literature of formal language and automata theory. In particular, as we explain in Section 5, our CEA are at the same time variants of Symbolic Finite State Automata [61] and Finite State Transducers [20]. These are established

variants of finite state automata to deal with infinite input alphabets and non-boolean output, respectively. A characteristic of CEA, distinguishing them from finite state transducers is that CEA produce complex events as output rather than strings over a finite alphabet. This makes the enumeration with output-linear delay while spending only constant time per input event challenging. (See Section 9.)

CEA do not support event correlation, nor computing aggregations over complex events. In recent years, automata models have been proposed that, while not yet applied to the CER domain, allow a form of event correlation or aggregation. In particular, register automata [48], data automata [22], and class memory automata [21] 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 [22] and class memory automata [21], 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. In an orthogonal line of research, cost register automata [11, 12] have been proposed as an extension of finite state automata that compute quantitative queries on data streams. These automata also operate on data words, but do not compare data values. Rather, they assume given a set of operations on the infinite set of data values D, and use these operations to output a value from D. For example, if one takes D to be the set of natural numbers and the set of operations to include addition, minimum and maximum, one can use cost register automata to compute sum, min, and max aggregates on the data values in D. Quantitative Regular Expressions [52] provide a user level language for cost register automata. Whether and how one can integrate the evaluation algorithm of CEA with the correlation capabilities of register and data automata and the aggregation capabilities of cost register automata is an interesting research question that we leave for future work.

Extensions of regular expressions with data filtering capabilities have been considered outside of the CER context. *Extended regular expressions* [7, 24, 25] extend the classical regular expressions operating on strings with variable binding expressions of the form $x\{e\}$ (meaning that when the input is matched, the substring matched by regular expression e is bound to variable x) and variable backreference expressions of the form $\&x$ (referring to the last binding of variable x). 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 [37]. 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.

CEL is a language in which variables bind to complex events. Since complex events are sets of primitive events, CEL is hence a language with second-order variables, similar to monadic second order logic (MSO) [49]. However, we are not aware of any language that combines regular operators with second-order variables as CEL, neither has it been proposed in the context of CER.

Finally, there has been some research in theoretical aspects of CER, e.g., in axiomatization of temporal models [62], privacy [46], and load shedding [45]. This literature does not study the semantics and evaluation of CER and, therefore, is orthogonal to our work.

type	H	T	H	H	T	T	T	H	H	...
id	2	0	0	1	1	0	1	1	0	...
value	25	45	20	25	40	42	25	70	18	
index	0	1	2	3	4	5	6	7	8	...

Fig. 1. A stream S of events measuring temperature and humidity. "value" contains degrees and humidity for T- and H- events, respectively.

3 EVENTS IN ACTION

We start by presenting the main features and challenges of CER. The examples used in this section will also serve throughout the paper as running examples.

In a CER setting, events arrive in a streaming fashion to a system that must detect certain *patterns* [32]. For the purpose of illustration 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 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. Fig. 1 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. Let us intuitively explain how we can express this as a pattern (also called a *formula*) in our framework:

$$\varphi_1 = (T\,;\,H)\ \texttt{FILTER}\ (T.tmp > 40\ \wedge H.hum <= 25\ \wedge T.id = 0\ \wedge H.id = 0).$$

This formula is asking for two events, one of type temperature (T) and one of type humidity (H), and the two events are filtered to select only those representing a high temperature followed by a low humidity measured by sensor 0. Event streams are typically noisy, and therefore one cannot expect in general that the T and H events of φ_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.

Whenever a pattern is detected in a stream, a corresponding *complex event* will be output. In our framework, each complex event will be the set of indices (stream positions) 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 φ_1 consists of sets $\{i, j\}$ such that $S[i]$ is of type T, $S[j]$ is of type H, $i < j$, and they satisfy the conditions expressed after the FILTER. By inspecting Fig. 1, we can see that the pairs satisfying these conditions are $\{1, 2\}$, $\{1, 8\}$, and $\{5, 8\}$.

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 φ_1' of φ_1 hence only returns $\{1, 2\}$ on the stream in Fig. 1.

$$\varphi_1' = (T : H)\ \texttt{FILTER}\ (T.tmp > 40\ \wedge H.hum <= 25\ \wedge T.id = 0\ \wedge H.id = 0).$$

Formulas φ_1 and φ_1' illustrate the two most elemental features of CER, namely *sequencing* (interpreted contiguously or not) and *local filtering* [3, 14, 23, 32, 64]. But although they detect a set of possible fires, they restrict the *order* in which the two events occur: the temperature must be measured before the humidity. Naturally, this could prevent the detection of a fire in which

the humidity was measured first. This motivates the introduction of *disjunction*, another common feature in CER engines [14, 32]. To illustrate, we extend φ_1 by allowing events to appear in arbitrary order.

$$\varphi_2 = [(T\,;\,H)\ \texttt{OR}\ (H\,;\,T)]\ \texttt{FILTER}\ (T.tmp > 40\ \wedge H.hum <= 25 \wedge T.id = 0\ \wedge H.id = 0)$$

The OR operator allows for any of the two patterns to be matched. The result of evaluating φ_2 over S (Fig. 1) is the same as the evaluation of φ_1 plus the complex event $\{2, 5\}$.

Because explicitly specifying all possible orders in which events may occur in a stream quickly becomes tedious (i.e., detecting n primitive events of distinct type, in any order, requires $n!$ permutations), there is often also a dedicated operator to express this more succinctly. In our framework, this operator is denoted ALL . The following formula φ_2' hence computes the same outputs as φ_2.

$$\varphi_2' = [T\ \texttt{ALL}\ H]\ \texttt{FILTER}\ (T.tmp > 40\ \wedge H.hum <= 25 \wedge T.id = 0\ \wedge H.id = 0).$$

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) [14, 32, 44] 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 [15, 34, 63]). Coming back to our example, the formula for measuring temperatures whenever an increase of humidity is detected by sensor 1 is:

$$\varphi_3 = [H\ \texttt{IN}\ H_1\,;\,T{+}\,;\,H\ \texttt{IN}\ H_2]\ \texttt{FILTER}\ (H_1.hum < 30\ \wedge H_2.hum > 60 \wedge H.id = 1 \wedge T.id = 1).$$

To understand the meaning of this formula, note that T and H refer to the types of events in the stream while H_1 and H_2 are *variables*. Intuitively, we use variables to represent complex events and *bind* them with the operator IN, to then filter them using predicates over complex events. This way variables H_1 and H_2 witness the increase of humidity from less than 30% to more than 60%, and T captures temperature measurements between H_1 and H_2. Strictly speaking H_1 and H_2 represent complex events, although because of the pattern each will always contain a single event. Note that T is not assigned to any variable in φ_3 despite that we later used T 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. Thus, T and H are also used as variables in φ_3.

Let us now explain the evaluation of φ_3 over S (Fig. 1). The only two humidity events satisfying the filter are $S[3]$ and $S[7]$, and the events in between that satisfy the filter over T are $S[4]$ and $S[6]$. Because + denotes *non-contiguous* iteration in CEL, the subformula $T+$ will match the complex events $\{4, 6\}$, $\{4\}$, and $\{6\}$. Hence, the output of φ_3 over S consists of the complex events $\{3, 4, 6, 7\}$, $\{3, 4, 7\}$ and $\{3, 6, 7\}$.

Similarly to the case for sequencing, some use cases require skipping of events under an iteration to be disallowed. For such situations CEL is also endowed with a *contiguous iteration* ($\oplus$). Let us denote by φ_3' the variant of φ_3 where + is replaced by $\oplus$. With contiguous iteration, the subformula $T\oplus$ matches only the complex events $\{4\}$ and $\{6\}$. It does not match the complex event $\{4, 6\}$ since $S[4]$ and $S[6]$ are not consecutive in the stream. As such, the output of φ_3' consists of complex events $\{3, 4, 7\}$ and $\{3, 6, 7\}$.

Formulas φ_3 and φ_3' output complex events that show how the temperature of sensor 1 changes when there is an increase of humidity. These complex events also contain the humidity-increase events themselves. If we wish to output complex events that contain only the temperature changes, without the humidity events, we may do so in our framework using the projection operator π:

$$\varphi_4 = \pi_T\big([H\ \texttt{IN}\ H_1\,;\,T{+}\,;\,H\ \texttt{IN}\ H_2]\ \texttt{FILTER}\ (H_1.hum < 30\ \wedge H_2.hum > 60 \wedge H.id = 1 \wedge T.id = 1)\big).$$

By projecting φ_3 on the variable T, φ_4 drops from the complex events in the output of φ_3 (namely: $\{3,4,6,7\}$, $\{3,4,7\}$ and $\{3,6,7\}$) all primitive events that were not matched by T. Hence, the output of φ_4 over S consists of the complex events $\{4,6\}$, $\{4\}$ and $\{6\}$.

The previous discussion raises an interesting question: are users interested in all complex events? Are some complex events more informative than others? Coming back to the output of φ_3 (namely: $\{3,6,7\}$, $\{3,4,7\}$ and $\{3,4,6,7\}$), one can argue that the largest complex event is more informative since all events are contained in it. Likewise, in the output of φ_1, some complex events that have the same second component (e.g., $\{1,8\}$ and $\{5,8\}$) represent a fire occurring at the same place and time, so one could argue that only one of the two is necessary. For cases like above, it is common to find CER systems that restrict the output by using so-called *selection strategies* (see for example [30, 63, 64]). Selection strategies are a fundamental feature of CER. Unfortunately, they have only been presented as heuristics applied to particular computational models, and thus their semantics are given by algorithms and are hard to generalize. A special mention deserves the *next* selection strategy (called skip-till-next-match in [63, 64]) which models the idea of outputting only those complex events that can be generated without skipping relevant events. Although the semantics of *next* has been mentioned in previous papers (e.g [19]), it is usually underspecified [63, 64] or complicates the semantics of other operators [34]. In Section 7, we formally define a set of selection strategies including *next*.

Before formally presenting our framework, we illustrate one more common feature of CER: *correlation*. Correlation is introduced by filtering events with predicates that involve more than one event. For example, consider that we want to see how does temperature change at some location whenever there is an increase of humidity, like in φ_3. What we need is a pattern where all the events are produced by the same sensor, but that sensor is not necessarily sensor 1. This is achieved by the following pattern:

$$\varphi_5 = [H \texttt{ IN } H_1 \,;\, T{+} \,;\, H \texttt{ IN } H_2] \texttt{ FILTER } (H_1.hum < 30 \ \wedge H_2.hum > 60 \wedge H.id = T.id)$$

Note that here the filter contains the predicate $H.id = T.id$ that force all events to have the same id.

4 A QUERY LANGUAGE FOR CER

In this section we formally introduce Complex Event Logic (CEL for short).

Schemas, Tuples and Streams. Let $\mathbf{A}$ be a set of *attribute names* and $\mathbf{D}$ 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 $\text{att}(R) \subseteq \mathbf{A}$. If R is a relation name, then an *R-tuple* is a function $t : \text{att}(R) \to \mathbf{D}$. For any relation name R, $\text{tuples}(R)$ denotes the set of all possible R-tuples[1], i.e., $\text{tuples}(R) = \{t : \text{att}(R) \to \mathbf{D}\}$. 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 [62]) to future work.

Let $\mathbf{L}$ be a finite set of *variables*. We assume that $\mathbf{L}$ contains all relation names (i.e., $\mathcal{R} \subseteq \mathbf{L}$). 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

[1]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.

$\varphi ::= R$	*R-tuple selection*
$\mid \varphi$ IN A	*Variable binding*
$\mid \varphi$ FILTER $P(\bar{A})$	*Filtering (both local & correlation)*
$\mid \varphi\,;\,\varphi$	*Non-contiguous sequencing*
$\mid \varphi+$	*Non-contiguous iteration*
$\mid \varphi : \varphi$	*Contiguous sequencing*
$\mid \varphi\oplus$	*Contiguous iteration*
$\mid \pi_L(\varphi)$	*Variable projection*
$\mid$ START(φ)	*Anchoring*
$\mid \varphi$ OR φ	*Disjunction*
$\mid \varphi$ AND φ	*Conjunction*
$\mid \varphi$ ALL φ	*Interleaved conjunction*
$\mid \varphi$ UNLESS φ	*Guarded negation*

Fig. 2. Syntax of CEL. R ranges over relation names, A over variables in $\mathbf{L}$, $P(\bar{A})$ over atoms over $\mathcal{P}$, and L over subsets of $\mathbf{L}$.

form $P(A_1, \ldots, A_n)$ where $P \in \mathcal{P}$ is a predicate of arity n, and $A_1, \ldots, A_n \in \mathbf{L}$. We also write $P(\bar{A})$ for $P(A_1, \ldots, A_n)$ when convenient.

CEL syntax and semantics. The syntax of CEL is given by the grammar in Fig. 2. There, R ranges over relation names, A over variables in $\mathbf{L}$, $P(\bar{A})$ over atoms over $\mathcal{P}$, and L over subsets of $\mathbf{L}$. It can readily be verified that all formulas introduced in Section 3 are CEL formulas. 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 [42] 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 $\mu : \mathbf{L} \to 2^{\mathbb{N}}$ that maps variables to complex events. The *support of* μ is the set of all positions appearing in complex events in the range of μ, $\sup(\mu) = \bigcup_{A \in \mathbf{L}} \mu(A)$. We denote by $A \mapsto C$ the valuation μ such that $\mu(A) = C$ and $\mu(B) = \emptyset$ for $B \in \mathbf{L} \setminus \{A\}$. For a valuation μ we denote by $\mu[A \mapsto C]$ the valuation μ' that equals μ on all variables except A, which it maps to C. Furthermore, if $L \subseteq \mathbf{L}$ we denote by $\mu|_L$ the restriction of μ to L: this is the valuation μ' such that $\mu'(A) = \mu(A)$ for every $A \in L$ while $\mu'(A) = \emptyset$ when $A \notin L$. We define the union between two valuations μ_1 and μ_2 by $(\mu_1 \cup \mu_2)(A) = \mu_1(A) \cup \mu_2(A)$ for every $A \in \mathbf{L}$.

$$
\begin{aligned}
\llbracket R \rrbracket(S,i,j) &= \{R \mapsto \{j\} \mid S[j] \in \text{tuples}(R)\},\\
\llbracket \varphi \texttt{ IN } A \rrbracket(S,i,j) &= \{\mu[A \mapsto \sup(\mu)]) \mid \mu \in \llbracket \varphi \rrbracket(S,i,j)\},\\
\llbracket \varphi \texttt{ FILTER } P(A_1,\ldots,A_n) \rrbracket(S,i,j) &= \{\mu \in \llbracket \varphi \rrbracket(S,i,j) \mid (S[\mu(A_1)],\ldots,S[\mu(A_n)]) \in P\},\\
\llbracket \varphi_1 \,;\, \varphi_2 \rrbracket(S,i,j) &= \{\mu_1 \cup \mu_2 \mid \exists k, i \le k < j : \mu_1 \in \llbracket \varphi_1 \rrbracket(S,i,k), \mu_2 \in \llbracket \varphi_2 \rrbracket(S,k+1,j)\},\\
\llbracket \varphi + \rrbracket(S,i,j) &= \llbracket \varphi \rrbracket(S,i,j) \cup \llbracket \varphi \,;\, \varphi + \rrbracket(S,i,j)\\
\llbracket \varphi_1 : \varphi_2 \rrbracket(S,i,j) &= \{\mu_1 \cup \mu_2 \mid \exists k, i \le k < j : \mu_1 \in \llbracket \varphi_1 \rrbracket(S,i,k), \mu_2 \in \llbracket \varphi_2 \rrbracket(S,k+1,j),\\
&\qquad \max(\sup(\mu_1)) = k, \min(\sup(\mu_2)) = k+1\},\\
\llbracket \varphi \oplus \rrbracket(S,i,j) &= \llbracket \varphi \rrbracket(S,i,j) \cup \llbracket \varphi : \varphi + \rrbracket(S,i,j)\\
\llbracket \pi_L(\varphi) \rrbracket(S,i,j) &= \{\mu|_L \mid \mu \in \llbracket \varphi \rrbracket(S,i,j)\}\\
\llbracket \texttt{ START}(\varphi) \rrbracket(S,i,j) &= \{\mu \in \llbracket \varphi \rrbracket(S,i,j) \mid \min(\sup(\mu)) = i\}\\
\llbracket \varphi_1 \texttt{ OR } \varphi_2 \rrbracket(S,i,j) &= \llbracket \varphi_1 \rrbracket(S,i,j) \cup \llbracket \varphi_2 \rrbracket(S,i,j)\\
\llbracket \varphi_1 \texttt{ 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 \texttt{ ALL } \varphi_2 \rrbracket(S,i,j) &= \{\mu_1 \cup \mu_2 \mid \exists i_1 \le j_1, i_2 \le j_2 : \mu_1 \in \llbracket \varphi_1 \rrbracket(S,i_1,j_1), \mu_2 \in \llbracket \varphi_2 \rrbracket(S,i_2,j_2),\\
&\qquad i = \min\{i_1,i_2\}, j = \max\{j_1,j_2\}\},\\
\llbracket \varphi_1 \texttt{ 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 every } i \le i' \le j' \le j\}
\end{aligned}
$$

Fig. 3. CEL semantics.

To define the semantics of a CEL formula φ, it is convenient to first define an auxiliary semantics $\llbracket \varphi \rrbracket$ that returns valuations, and to then define the (final) semantics $[\![\varphi]\!]$ that returns complex events based on this auxiliary semantics. Concretely, the auxiliary valuation semantics of φ over a stream S, starting at position i and ending at position $j \ge i$, denoted $\llbracket \varphi \rrbracket(S,i,j)$, is defined by induction on the structure of φ as shown in Fig. 3.

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

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

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 φ.

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 $[\![\varphi]\!](S,0,n)$ by $[\![\varphi]\!]_n(S)$.

Notational conventions. Throughout the paper we use φ FILTER $(P(\bar{A}) \wedge Q(\bar{B}))$ or φ FILTER $(P(\bar{A}) \vee Q(\bar{B}))$ as syntactic sugar for $(\varphi \texttt{ FILTER } P(\bar{A}))$ FILTER $Q(\bar{B})$ and $(\varphi \texttt{ FILTER } P(\bar{A}))$ OR $(\varphi \texttt{ FILTER } Q(\bar{B}))$, 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 $P = \{A \subseteq \text{tuples}(\mathcal{R}) \mid \forall t \in A\colon t(x) = c\}$. 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 $\{(H,T) \in 2^{\text{tuples}(\mathcal{R})} \times 2^{\text{tuples}(\mathcal{R})} \mid \forall h \in H, \forall t \in T, t(id) = h(id)\}$.

Examples. Under these conventions, it can be verified that the formulas introduced in Section 3, when executed over the stream in Fig. 1, indeed yield the complex events claimed in Section 3.

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 φ_1 to be consecutive with the minimal position in the complex event output by φ_2. Because $\oplus$ is defined through repeated application of $:$, this implies that

a formula $\varphi\oplus$ does not impose that all events matched by φ 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.

Discussion of language features. As discussed in the Introduction, Table 1 lists a set of CER operators that, according to recent surveys in the field [10, 39], can be considered "basic operators that should be present in every CER language" [39]. If we relate CEL to the operators mentioned in this table, then it is clear that CEL includes sequencing (both contiguous and non-contiguous), disjunction, iteration (contiguous and non-contiguous), filtering (with arbitrary predicates, hence supporting both local filtering and correlation), conjunction (two forms), and negation. The last three operators of Table 1 require more discussion.

While CEL does have a projection operator, this operator projects *valuations*. By contrast, the projection operator referred to in Table 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 1, one could specify that only the *id* attribute of the stream in Fig. 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 CEL does not have an explicit windowing operator, windowing is expressible using filtering. For example, assume that we wish to evaluate formula φ_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.,

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

then evaluation of φ_1 over such a sliding window is expressed by $\varphi_1'' = (\varphi_1 \texttt{ IN } W) \texttt{ FILTER } Q(W)$.

Selection strategies operators for CEL will be introduced in Section 7.

An operator of CEL that does not occur in Table 1, is the anchoring operator `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 5. Actually, this operator is intensively used in the context of regular expression programing where an expression of the form "$\hat{}R$" 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.

Unary CEL. As discussed in the Introduction, 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. Indeed, formulas like φ_1, φ_1', φ_2, φ_2', φ_3, φ_3', and φ_4 of Section 3 use only local filters and one can verify, for any position in the stream, whether there exists a complex event that satisfies the formula using only a bounded amount of memory. Hence, these formulas can be considered "regular", in similarity to the rich theory of regular formal languages, which can also be recognized using bounded memory. In contrast, formulas that express correlation, such as φ_5 of Section 3, are intuitively more complex. For φ_5 in particular, the id of the events must be remembered in order to compare it with future incoming events, suggesting that more memory is required. Of course, one would like to have a full-fledged model that applies to any CEL formula, but to this end we wish to first understand the regular fragment.

We will refer to the regular formulas of CEL as *unary CEL* formulas, whose formal definition is as follows. A *CEA predicate* is any decidable subset of tuples($\mathcal{R}$). CEA 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 the paper we assume given a fixed universe $\mathcal{U}$ of CEA predicates, which contains at least the empty predicate $\emptyset$; and for every relation symbol $R \in \mathcal{R}$, the predicate 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 CEA predicate, then we denote by P^{SO} its *second order* extension, which is the CEL predicate such that $P^{\text{SO}} = \{S \subseteq \text{tuples}(\mathcal{R}) \mid P(t) \text{ for every } t \in S\}$. We extend this definition to sets of CEA predicates: if $\mathcal{P}$ is a set of CEA predicates, then $\mathcal{P}^{\text{SO}}$ is the set $\{P^{\text{SO}} \mid P \in \mathcal{P}\}$.

Definition 4.1. A CEL formula φ is *unary* if for every subformula of the form φ' FILTER $P(\bar{A})$, it holds that $P(\bar{A})$ is the second order extension of a CEA predicate (i.e. $P(\bar{A}) \in \mathcal{U}^{\text{SO}}$).

For example, formulas φ_1, φ_1', φ_2, φ_2', φ_3, φ_3', and φ_4 in Section 3 are unary. Indeed, if we decompose the $\wedge$ and $\vee$ boolean connectives used inside filters in these formulas into atomic predicates, we see that all the atomic predicates are unary. In contrast, the formula φ_5 is not unary: the CEL predicate $H.id = T.id$ is a binary CEL predicate, and hence not the second order extension of a CEA predicate. Also, the formula φ_1'' defined above, which evaluates φ_1' over a sliding window of 100 events, is not unary since Q compares the first and last event occurring in a complex event. We remark that windowing is not expressible in unary CEL in general.

5 A COMPUTATIONAL MODEL FOR CEL

In this section, we introduce a formal computational model for evaluating CEL formulas called *complex event automata* (CEA for short). Specifically, CEA are designed to be an evaluation model for unary CEL.

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 [61], 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* [20], 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 4 that $\mathcal{U}$ denotes our universe of CEA predicates which contains at least the empty predicate $\emptyset$; and for every relation symbol $R \in \mathcal{R}$, the predicate tuples(R); and which is closed under union, intersection and difference. As a consequence, $\mathcal{U}$ also contains the predicate tuples($\mathcal{R}$) $= \cup_{R \in \mathcal{R}}$ tuples(R), which we will sometimes simply denote by TRUE for clarity and emphasis. Recall that $\mathbf{L}$ is a set of variables.

Definition 5.1. A *complex event automaton* (CEA) is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of states, $\Delta \subseteq Q \times \mathcal{U} \times 2^{\mathbf{L}} \times Q$ 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* $|\mathcal{A}|$ of $\mathcal{A}$ is its number of states plus its number of edges, $|\mathcal{A}| = |Q| + |\Delta|$. Given a stream $S = t_0 t_1 \ldots$, a *run* ρ *of* $\mathcal{A}$ *over* S is a sequence of

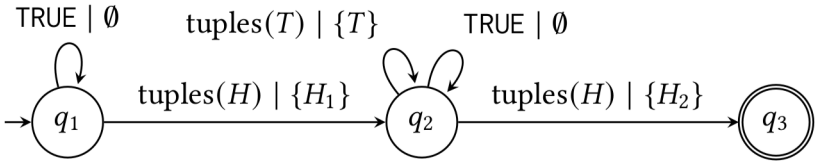

Fig. 4. A CEA that can generate an unbounded amount of complex events.

transitions: $\rho : q_0 \xrightarrow{P_0/L_0} q_1 \xrightarrow{P_1/L_1} \cdots \xrightarrow{P_n/L_n} q_{n+1}$ 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 ρ is *accepting* if $q_{n+1} \in F$. A position $0 \leq i \leq n$ is said to be *marked by variable* $A \in \mathbf{L}$ in ρ 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 $L \subseteq \mathbf{L}$ 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 μ_ρ such that, for every variable A, $\mu_\rho(A)$ is the complex event consisting of all positions marked by A in ρ, $\mu_\rho(A) = \{0 \leq i \leq n \mid A \in L_i\}$. The complex event associated to ρ is the set of all positions marked by ρ.

Definition 5.2. Let $\mathcal{A}$ be a CEA and S a stream. Let $\text{Run}_n(\mathcal{A}, S)$ denote the set of all accepting runs of $\mathcal{A}$ over S that end at position $n \in \mathbb{N}$. The set of *valuations of* $\mathcal{A}$ *over* S *at position* n is defined as $[\![\mathcal{A}]\!]_n(S) = \{\mu_\rho \mid \rho \in \text{Run}_n(\mathcal{A}, S)\}$. The set of all *complex events recognized by* $\mathcal{A}$ *over* S *at position* n is defined as $[\![\mathcal{A}]\!]_n(S) = \{\sup(\mu) \mid \mu \in [\![\mathcal{A}]\!]_n(S)\}$.

Example 5.3. Consider as an example the CEA $\mathcal{A}$ depicted in Fig. 4. In this CEA, the transition $(q_2, \text{tuples}(T) \,|\, \{T\}, q_2)$ 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 $\text{TRUE} \,|\, \emptyset$ allow $\mathcal{A}$ to arbitrarily skip tuples of the stream. Then, for every stream S, $[\![\mathcal{A}]\!]_n(S)$ 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 [20] 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 9. 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.

I/O-deterministic CEA. Just like standard FSA, one may distinguish between deterministic and non-deterministic CEA, in the following sense.

Definition 5.4. 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 μ, there is at most one run over S that generates μ (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.4 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 9.

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 $[\![\mathcal{A}_1]\!]_n(S) = [\![\mathcal{A}_2]\!]_n(S)$. 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 5.5. *The class of CEA is closed under I/O-determinization: for every CEA $\mathcal{A}$ there is a valuation-equivalent I/O-deterministic CEA $\mathcal{A}'$ whose size is at most exponential in $|\mathcal{A}|$.*

The proof requires the following notion.

Definition 5.6. 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 5.5.

Proof of Proposition 5.5. The construction is similar to the one used for classical NFA determinization: $\mathcal{A}'$ will maintain, in its states, the *set* of states that $\mathcal{A}$ 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 $\mathcal{A}$—all with different predicates—that can read t. How to encode all of these transitions by single transition—and hence one predicate—in $\mathcal{A}'$? 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 $\mathcal{A}$ could take when reading t.

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

$$\Delta(R, P, L) = \{q \in Q \mid \text{exist } p \in R \text{ and } P' \in \mathcal{P} \text{ such that } P \subseteq P' \text{ and } (p, (P', L), q) \in \Delta\},$$

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 a accepting run in $\mathcal{A}'$ where the set-states contain the states from the original run. On the

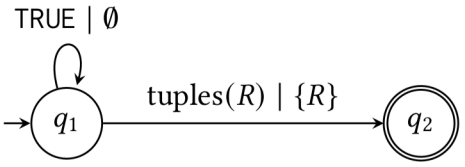

Fig. 5. A CEA for the atomic formula R.

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 types($\mathcal{P}$) we have:

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

The size of $\mathcal{A}'$ is hence exponential in $|\mathcal{A}|$. □

6 COMPARING UNARY CEL AND CEA

In this section, we show that unary CEL and CEA are expressively equivalent. We first show how to translate unary CEL into CEA, and then conversely how to translate CEA into unary CEL. As a by-product of these translations, we obtain that several operators are not primitive in unary CEL, and can be expressed using only variable binding, filtering, contiguous sequencing and iteration, disjunction, projection, and anchoring.

Definition 6.1. A CEL formula φ is *valuation equivalent* with CEA $\mathcal{A}$ if $[\![\varphi]\!]_n(S) = [\![\mathcal{A}]\!]_n(S)$ for every S and n. They are *complex event equivalent* if $[\![\varphi]\!]_n(S) = [\![\mathcal{A}]\!]_n(S)$ for every S and n.

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

Theorem 6.2. *For every unary CEL formula φ there is a valuation-equivalent CEA $\mathcal{A}_\varphi$.*

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 $S = t_0 t_1 \ldots$ be a stream and $\mathcal{A} = (Q, \Delta, I, F)$ be a CEA. For a valuation μ and $i \in \mathbb{N}$ define the valuation μ^{+i} such that $\mu^{+i}(A) = \{k + i \mid k \in \mu(A)\}$ for every $A \in \mathbf{L}$. Recall that $S_i = t_i t_{i+1} \ldots$. Then we define the set of valuations of $\mathcal{A}$ over S from positions i to j as $[\![\mathcal{A}]\!](S, i, j) = \{\mu^{+i} \mid \mu \in [\![\mathcal{A}]\!]_{j-i}(S_i)\}$. In other words, $[\![\mathcal{A}]\!](S, i, j)$ computes all the valuations obtained from the evaluation of $\mathcal{A}$ from positions i to j.

To obtain the theorem we prove the following stronger property: For every unary CEL formula φ there exists a CEA $\mathcal{A}_\varphi$ such that

$$[\![\varphi]\!](S, i, j) = [\![\mathcal{A}_\varphi]\!](S, i, j), \text{ for every stream } S \text{ and positions } i \leq j. \tag{1}$$

The construction of $\mathcal{A}_\varphi$ is by induction on φ.

- If $\varphi = R$, then $\mathcal{A}_\varphi$ is defined as depicted in Fig. 5, i.e. $\mathcal{A}_\varphi = (\{q_1, q_2\}, \Delta_\varphi, \{q_1\}, \{q_2\})$ with $\Delta_\varphi = \{(q_1, \text{tuples}(R), \{R\}, q_2), (q_1, \mathsf{TRUE}, \emptyset, q_1)\}$.
- If $\varphi = \psi$ IN A, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where Δ_φ is the result of adding variable A to all marking transitions of Δ_ψ. Formally, $\Delta_\varphi = \{(p, P, L, q) \in \Delta_\psi \mid L = \emptyset\} \cup \{(p, P, L, q) \mid \exists L' \neq \emptyset$ such that $(p, P, L', q) \in \Delta_\psi \wedge L = L' \cup \{A\}\}$.
- If $\varphi = \psi$ FILTER $P^{\text{SO}}(A)$ 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 Δ_φ is defined as $\{(p, P', L, q) \in \Delta_\psi \mid A \notin L\} \cup \{(p, P \wedge P', L, q) \mid$

$(p, P', L, q) \in \Delta_\psi \wedge A \in L\}$. 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 $\varphi = \psi_1 ; \psi_2$, then $\mathcal{A}_\varphi = (Q_{\psi_1} \cup Q_{\psi_2}, \Delta_\varphi, I_{\psi_1}, F_{\psi_2})$ where $\Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \cup \{(p, P, L, q) \mid q \in I_{\psi_2} \wedge \exists q' \in F_{\psi_1}.(p, P, L, q') \in \Delta_{\psi_1}\}$. 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 $\Delta_\varphi = \Delta_\psi \cup \{(p, P, L, q) \mid q \in I_\psi \wedge \exists q' \in F_\psi.(p, P, L, q') \in \Delta_\psi\}$.
- If $\varphi = \psi_1 : \psi_2$, then we do the following. In order to obtain the contiguous sequencing semantics, we will we 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 $\mathcal{A}_\varphi$ that generates a valuation μ will be able to be decomposed into accepting runs of $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ generating valuations μ_1 and μ_2 with $\max(\sup(\mu_1)) + 1 = \min(\sup(\mu_2))$. Formally, we define $\mathcal{A}_\varphi = (Q_\varphi, \Delta_\varphi, I_\varphi, F_\varphi)$ as follows. First, the set of states is $Q_\varphi = Q_{\psi_1} \cup Q_{\psi_2} \cup \{q\}$, where q is a new fresh state. Then, the transition relation is $\Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \cup \{(q_1, P, L, q) \mid L \neq \emptyset \wedge \exists q' \in F_{\psi_1}.\ ((q_1, P, L, q') \in \Delta_{\psi_1})\} \cup \{(q, P, L, q_2) \mid L \neq \emptyset \wedge \exists q' \in I_{\psi_2}.\ ((q', P, L, q_2) \in \Delta_{\psi_2})\}$. 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 $\Delta_\varphi = \Delta_\psi \cup \{(q_1, P, L, q) \mid L \neq \emptyset \wedge \exists q' \in F_\psi.\ ((q_1, P, L, q') \in \Delta_\psi)\} \cup \{(q, P, L, q_2) \mid L \neq \emptyset \wedge \exists q' \in I_\psi.\ ((q', P, L, q_2) \in \Delta_\psi)\}$.
- If $\varphi = \pi_L(\psi)$ for some $L \subseteq \mathbf{L}$, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where Δ_φ is the result of intersecting the labels of each transition in Δ_ψ with L. Formally, that is $\Delta_\varphi = \{(p, P, L \cap L', q) \mid (p, P, L', q) \in \Delta_\psi\}$.
- If $\varphi = \texttt{START}(\psi)$, 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 $\Delta_\varphi = \Delta_\psi \cup \{(q, P, L, p) \mid L \neq \emptyset \wedge \exists q' \in I_\psi.((q', P, L, p) \in \Delta_\psi)\}$.
- If $\varphi = \psi_1 \texttt{ OR } \psi_2$, 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 $\varphi = \psi_1 \texttt{ AND } \psi_2$, 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 $\Delta_\varphi = \{((q_1, q_2), P_1 \cap P_2, L, (p_1, p_2)) \mid (q_1, P_1, L, p_1) \in \Delta_{\psi_1}, (q_2, P_2, L, p_2) \in \Delta_{\psi_2}\}$.
- If $\varphi = \psi_1 \texttt{ ALL } \psi_2$, 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 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 Δ_φ is defined as follows.

$$\begin{aligned}\Delta_\varphi = &\{((q_1,q_2), P_1 \cap P_2, L_1 \cup L_2, (p_1,p_2)) \mid (q_1,P_1,L_1,p_1) \in \Delta_{\psi_1}, (q_2,P_2,L_2,p_2) \in \Delta_{\psi_2}\}\\ &\cup \{((q_1,q_2), P_2, L_2, (q_1,p_2)) \mid q_1 \in I_{\psi_1}, (q_2,P_2,L_2,p_2) \in \Delta_{\psi_2}\}\\ &\cup \{((q_1,q_2), P_1, L_1, (p_1,q_2)) \mid q_2 \in I_{\psi_2}, (q_1,P_1,L_1,p_1) \in \Delta_{\psi_1}\}\\ &\cup \{((q_1,q_2), P_2, L_2, (q_1,p_2)) \mid q_1 \in F_{\psi_1}, (q_2,P_2,L_2,p_2) \in \Delta_{\psi_2}\}\\ &\cup \{((q_1,q_2), P_1, L_1, (p_1,q_2)) \mid q_2 \in F_{\psi_2}, (q_1,P_1,L_1,p_1) \in \Delta_{\psi_1}\}.\end{aligned}$$

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 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 $\varphi = \psi_1$ UNLESS ψ_2 then we first construct the automaton $\mathcal{E} = (Q_\mathcal{E}, \Delta_\mathcal{E}, I_\mathcal{E}, F_\mathcal{E})$ for the formula $\psi_2' = \pi_\emptyset(\psi_2)$, 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, $\mathcal{A}_\varphi = (Q_{\psi_1} \times Q_\mathcal{E}, \Delta_\varphi, I_{\psi_1} \times I_\mathcal{E}, F_{\psi_1} \times (Q_\mathcal{E} \setminus F_\mathcal{E}))$ where $\Delta_\varphi = \{((q_1,q_2), P_1 \cap P_2, L, (p_1,p_2)) \mid (q_1,P_1,L,p_1) \in \Delta_{\psi_1}, (q_2,P_2,\emptyset,p_2) \in \Delta_\mathcal{E}\}$.

Property (1) is now obtained by induction on φ. □

Theorem 6.3. *For every CEA $\mathcal{A}$, there exists a valuation-equivalent unary CEL formula $\varphi_\mathcal{A}$ that does not use any operator in* { ; , +, AND , ALL , UNLESS }.

Proof. Let $\mathcal{A} = (Q, \Delta, I, F)$ 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 φ_{ij} that represents the complex events defined by the runs from q_i to q_j. To aid in the definition of φ_{ij} we will first define, for every $1 \leq k \leq |Q|$ the formula φ_{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 φ_{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 $\varphi_{ij}^0 =$ FALSE where FALSE is a formula that is never satisfied. One way to define it is FALSE = (R FILTER $\emptyset$). If there is at least one transition from q_i to q_j in $\mathcal{A}$, we do the following. For convenience, if $L = \{A_1, \ldots, A_l\}$ is a non-empty set of variables and ψ is a CEL

formula, let us simply write ψ IN L for the more verbose $(\ldots((\psi$ IN $A_1)$ IN $A_2)$ IN $\ldots)$ IN A_l. Assume that $\mathcal{R} = \{R_1, \ldots, R_r\}$. Define, for any transition $t = (q_i, P, L, q_j)$ of $\mathcal{A}$ the formula ψ_t by

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

where A is a "fresh" variable, distinct from the relation names in $\mathcal{R}$ and any variable mentioned in $\mathcal{A}$. Intuitively, ψ_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

$$\varphi_{ij}^0 = \psi_{t_1} \text{ OR } \ldots \text{ OR } \psi_{t_m},$$

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:

$$\varphi_{ij}^k \;=\; \varphi_{ij}^{k-1} \text{ OR } (\varphi_{ik}^{k-1} : \varphi_{kj}^{k-1}) \text{ OR } (\varphi_{ik}^{k-1} : \varphi_{kk}^{k-1} \oplus : \varphi_{kj}^{k-1})$$

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

By combining theorems 6.2 and 6.3 we obtain that { ; , +, AND , ALL , UNLESS } are expressively redundant in unary CEL.

Corollary 6.4. *For every unary CEL formula there exists a valuation-equivalent CEL formula constructed only from the operators in* { IN , FILTER , : , $\oplus$, π, START, OR }.

7 SELECTION STRATEGIES

Matching complex events is a computationally intensive task, especially when non-contiguous sequencing and iteration are used. Indeed, as the examples in Section 3 suggest, non-contiguous sequencing and iteration can cause the amount of generated complex events to grow exponentially in the size of the stream. Not all of the produced complex events may actually be useful for the user. For instance, reconsider formula φ_1 from our running example (Section 3). 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, φ_1 outputs complex events $\{1, 2\}$, $\{1, 8\}$, and $\{5, 8\}$ when evaluated on stream S of Fig.1. 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 succesfully 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*) [32] to a CER pattern to restrict the set of results [26, 63, 64] 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 [44], 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 selection strategies can be compiled into CEA, whose expressive power is equivalent to unary CEL.

7.1 The semantics of common selection strategies in CER

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 φ_1 in Section 3 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 preprocessed 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 complex events. Formally, for any CEL formula φ we have that $\mu \in [\![\texttt{STRICT}(\varphi)]\!](S, i, j)$ holds if $\mu \in [\![\varphi]\!](S, i, j)$ 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, $\texttt{STRICT}(\varphi_1)$ would only produce $\{1, 2\}$, although $\{1, 8\}$ and $\{5, 8\}$ are also outputs for φ_1 over S. In other words, $\texttt{STRICT}(\varphi_1)$ is equivalent to φ_1'. We will have more to say about the relationship between STRICT and the contiguous sequencing (:) and iteration ($\oplus$) operators in Section 8.2.

NXT. The second selector, NXT, is similar to the *skip-till-next-match* operator proposed in [44]. 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 satisfying the next element of the query. In [44] 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 7.1. *$\leq_{\text{next}}$ is a total order between complex events.*

The proof of Lemma 7.1 is technical and the reader may find it in the Appendix.

We can define now the semantics of NXT: for a CEL formula φ we have $\mu \in [\![\texttt{NXT}(\varphi)]\!](S, i, j)$ if $\mu \in [\![\varphi]\!](S, i, j)$ and $\sup(\mu') \leq_{\text{next}} \sup(\mu)$ for every $\mu' \in [\![\varphi]\!](S, i, j)$. 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 $[\![\texttt{NXT}(\varphi_1)]\!]_8(S)$ but $\{5, 8\}$ is not. Furthermore, $\{3, 4, 6, 7\}$ is in $[\![\texttt{NXT}(\varphi_3)]\!]_7(S)$ while $\{3, 4, 7\}$ and $\{3, 6, 7\}$ are not.

LAST. The 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 φ_1 and stream S (Section 3), we know that every complex event in $\mathsf{NXT}(\varphi_1)$ will have event 1. In this sense, the 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 LAST. Formally, the LAST selector is defined exactly as 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, $\mathsf{LAST}(\varphi_1)$ would select the *most recent* temperature and humidity that explain the matching of φ_1 (i.e. $\{5, 8\}$), which might be a better explanation for a possible fire. Surprisingly, as we will see LAST enjoys the same computational properties as NXT, even though it does not come from a greedy heuristic like NXT does.

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 φ we say that $\mu \in [\![\mathsf{MAX}(\varphi)]\!](S, i, j)$ holds iff $\mu \in [\![\varphi]\!](S, i, j)$ and for all $\mu' \in [\![\varphi]\!](S, i, j)$, 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 φ_1, the 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 φ_3 produced $\{3, 6, 7\}$, $\{3, 4, 7\}$, and $\{3, 4, 6, 7\}$. Then, $\mathsf{MAX}(\varphi_3)$ 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 $\mathsf{NXT}(\varphi_3)$ and $\mathsf{LAST}(\varphi_3)$ over the stream we will also get $\{3, 4, 6, 7\}$ as the only output, illustrating that NXT and LAST also yield complex events with maximal information.

7.2 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., 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 6.3, 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 9).

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 $\mathsf{SEL} \in \{\mathsf{STRICT}, \mathsf{NXT}, \mathsf{LAST}, \mathsf{MAX}\}$ and stream S, the set of valuations $[\![\mathsf{SEL}(\mathcal{A})]\!]_n(S)$ is defined analogously to $[\![\mathsf{SEL}(\varphi)]\!]_n(S)$ for a formula φ. Then, we say that a CEA $\mathcal{A}_1$ is valuation equivalent to $\mathsf{SEL}(\mathcal{A}_2)$ if $[\![\mathcal{A}_1]\!]_n(S) = [\![\mathsf{SEL}(\mathcal{A}_2)]\!]_n(S)$ for every stream S and position n.

Theorem 7.2. *Let* SEL *be a selection strategy. For any CEA $\mathcal{A}$, there is a CEA $\mathcal{A}_{\mathsf{SEL}}$ that is valuation equivalent to* $\mathsf{SEL}(\mathcal{A})$. *Furthermore, $\mathcal{A}_{\mathsf{SEL}}$ 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 9, 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 Δ. Moreover, let $\mathcal{L}$ be the set of all variables sets used in Δ. For the next constructions, we extend the transition relation Δ as a function:

$$\Delta(R, P, L) = \{q \in Q \mid \text{exist } p \in R \text{ and } P' \in \mathcal{P} \text{ such that } P \subseteq P' \text{ and } (p, P', L, q) \in \Delta\}$$

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

`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 $\mathcal{A}_{\mathsf{STRICT}} = (Q_{\mathsf{STRICT}}, \Delta_{\mathsf{STRICT}}, I_{\mathsf{STRICT}}, F_{\mathsf{STRICT}})$ component by component. First, we define the set of states as $Q_{\mathsf{STRICT}} = 2^Q \times \{1, 2, 3\}$. 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 $\Delta_{\mathsf{STRICT}} = \{((R, i), P, L, (R', i')) \mid i, L, i' \text{ preserve an interval and } R' = \Delta(R, P, L)\}$. Finally, the sets of initial and final states are $I_{\mathsf{STRICT}} = \{(I, 1)\}$ and $F_{\mathsf{STRICT}} = \{(R, i) \mid R \cap F \neq \emptyset \text{ and } i \in \{1, 2, 3\}\}$.

This construction maintains the following invariant: for every run ρ of $\mathcal{A}$ over S ending at some state q, there is a run ρ' of $\mathcal{A}_{\mathsf{STRICT}}$ 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 $\mathcal{A}_{\mathsf{STRICT}}$ 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 $\mathcal{A}_{\mathsf{STRICT}}$ is of size at most linear.

`NXT` *operator.* We define $\mathcal{A}_{\mathsf{NXT}} = (Q_{\mathsf{NXT}}, \Delta_{\mathsf{NXT}}, I_{\mathsf{NXT}}, F_{\mathsf{NXT}})$ where $Q_{\mathsf{NXT}} = 2^Q \times 2^Q$, Δ_{NXT} is as described below, $I_{\mathsf{NXT}} = \{(I, \emptyset)\}$, and $F_{\mathsf{NXT}} = \{(R, W) \mid R \cap F \neq \emptyset \wedge W \cap F = \emptyset\}$.

We say that a run ρ_1 of $\mathcal{A}$ *wins* over a run ρ_2 of $\mathcal{A}$ if $\sup(\mu_{\rho_2}) <_{\text{next}} \sup(\mu_{\rho_1})$. Intuitively, we will define the transition relation of $\mathcal{A}_{\mathsf{NXT}}$ in such a way that if a run ρ of $\mathcal{A}_{\mathsf{NXT}}$ reaches state (R, W) of Q_{NXT}, then R represents a set of "current" states of $\mathcal{A}$, i.e., those for which there is a run reaching them that has the same output as ρ. Importantly, W will be the set of states of $\mathcal{A}$ having runs that win over the current run ρ. As such, if W contains a final state, then μ_ρ should not be output by $\mathcal{A}_{\mathsf{NXT}}$.

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

$$\Delta_{\bar{\emptyset}} = \{((R, W), P, L, (R', W')) \mid W' = \Delta(W, P) \wedge R' = \Delta(R, P, L) \setminus W'\},$$
$$\Delta_{\emptyset} = \{((R, W), P, \emptyset, (R', W')) \mid W' = \Delta(W, P) \cup \big(\cup_{L \in \mathcal{L} \setminus \{\emptyset\}} \Delta(R, P, L)\big) \wedge R' = \Delta(R, P, \emptyset) \setminus W'\}.$$

where P ranges over all predicates on $\text{types}(\mathcal{P})$ and L over all variable sets on $\mathcal{L} \setminus \{\emptyset\}$. Note that, from the extension of Δ to sets of states, it is implicitly required that $P \in \text{types}(\mathcal{P})$ (recall the definition and construction of $\text{types}(\mathcal{P})$ for the proof of Proposition 5.5). Then, $\Delta_{\mathsf{NXT}} = \Delta_{\bar{\emptyset}} \cup \Delta_{\emptyset}$. 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 `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 $\mathcal{A}_{\mathsf{NXT}}$ is I/O-deterministic and that it is complete. Indeed, given a state $(R, S) \in Q_{\mathsf{NXT}}$ and a mark L, for any tuple t there is exactly one transition $((R, W), P, L, (R', W')) \in \Delta_{\mathsf{NXT}}$ such that $t \in P$. Further, consider a stream $S = t_0 t_1 \dots$ and marks $L_0, L_1, \dots, L_n$. Then, there is a run of $\mathcal{A}_{\mathsf{NXT}}$ 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:

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

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

Now, we analyse the size of the automaton $\mathcal{A}_{\mathsf{NXT}}$. First, one can see that $|Q_{\mathsf{NXT}}|$ is bounded by $2^{2|Q|}$, 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 $|\Delta_{\mathsf{NXT}}| \leq |Q_{\mathsf{NXT}}| \cdot 2 \cdot 2^{|\Delta|}$. Therefore, the resulting automaton $\mathcal{A}_{\mathsf{NXT}}$ is of size at most exponential over the size of $\mathcal{A}$.

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 $\mathcal{A}_{\mathsf{LAST}} = (Q_{\mathsf{LAST}}, \Delta_{\mathsf{LAST}}, I_{\mathsf{LAST}}, F_{\mathsf{LAST}})$ where $Q_{\mathsf{LAST}} = 2^Q \times 2^Q \times 2^Q$, $I_{\mathsf{LAST}} = (I, \emptyset, \emptyset)$ and $F_{\mathsf{LAST}} = \{(R, W, T) \mid R \cap F \neq \emptyset \wedge W \cap F = \emptyset\}$. For the following, we say that a run ρ_1 *wins* over a run ρ_2 if $\sup(\mu_{\rho_2}) <_{\text{last}} \sup(\mu_{\rho_1})$. If a run ρ of $\mathcal{A}_{\mathsf{LAST}}$ reaches a state $(R, W, T) \in Q_{\mathsf{LAST}}$, R represents a set of current states of $\mathcal{A}$, W is a set of states of $\mathcal{A}$ having runs that win over ρ, and T is a set of states of $\mathcal{A}$ having runs such that ρ 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, $\Delta_{\bar{\emptyset}}$, consists of all tuples $((R, W, T), P, L, (R', W', T'))$ such that $W' = \cup_{L \in \mathcal{L} \setminus \{\emptyset\}} \Delta(W, P, L)$, $R' = \Delta(R, P, L) \setminus W'$ 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 $W' = \Delta(W, P) \cup \big(\cup_{L \in \mathcal{L} \setminus \{\emptyset\}} \Delta(R \cup T, P, L)\big)$, $R' = \Delta(R, P, \emptyset) \setminus W'$ and $T' = \Delta(T, P, \emptyset) \setminus (R' \cup W')$. Then, $\Delta_{\mathsf{LAST}} = \Delta_{\bar{\emptyset}} \cup \Delta_\emptyset$.

One can derive an explanation similar to the one in the 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 ρ_1, ρ_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 ρ_i and let ρ_i' be ρ_i without its last transition. Now, let us consider the following tree of cases regarding events(ρ_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 $\sup(\mu_{\rho_2'}) <_{\text{last}} \sup(\mu_{\rho_1'})$ 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 Δ_{LAST}.

Like in the previous construction, here the resulting automaton is I/O-deterministic and complete. Moreover, for any stream $S = t_0 t_1 \dots$ and marks $L_0, L_1, \dots L_n$, there is a run of $\mathcal{A}_{\mathsf{LAST}}$ over S:

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

that satisfies the following invariants:

(1) W_{n+1} is the set of all $q \in Q$ for which there is a run ρ' of $\mathcal{A}$ over S at position n ending at q and $\sup(\mu_\rho) <_{\text{last}} \sup(\mu_{\rho'})$,
(2) R_{n+1} is the set of all $q \in Q$ for which there is a run ρ' of $\mathcal{A}$ over S at position n ending at q with $\mu_\rho = \mu_{\rho'}$, minus W_{n+1}, and
(3) T_{n+1} is the set of all $q \in Q$ for which there is a run ρ' of $\mathcal{A}$ 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 ρ 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 $\mathcal{A}$. By setting this as our accepting states, it follows that $\mathcal{A}_{\mathsf{LAST}}$ is equivalent to $\mathsf{LAST}(\mathcal{A})$. Moreover, one can check that the size of $\mathcal{A}_{\mathsf{LAST}}$ is at most exponential over the size of $\mathcal{A}$.

`MAX` *operator.* For this case, we use a similar construction as for `NXT`, maintaining only two sets of states in each state of the resulting CEA. Formally, we define $\mathcal{A}_{\mathsf{MAX}} = (Q_{\mathsf{MAX}}, \Delta_{\mathsf{MAX}}, I_{\mathsf{MAX}}, F_{\mathsf{MAX}})$ with $Q_{\mathsf{MAX}} = 2^Q \times 2^Q$, $I_{\mathsf{MAX}} = \{(I, \emptyset)\}$ and $F_{\mathsf{MAX}} = \{(R, W) \mid R \cap F \neq \emptyset \wedge W \cap F = \emptyset\}$. Again, for each $(R, W) \in Q_{\mathsf{MAX}}$, 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:

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

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

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

8 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.

To obtain this understanding we consider the following fragment of CEL, composed of only the most fundamental operators, with sequencing and iteration restricted to be non-contiguous.

Definition 8.1. Define CEL* to be the fragment of CEL consisting of formulas that can be built from relation variables using only variable binding, filtering, non-contiguous sequencing, non-contiguous iteration, and disjunction, as given by the following grammar:

$$\varphi \ ::= \ R \mid \varphi \ \mathtt{IN} \ A \mid \varphi \ \mathtt{FILTER} \ P(\bar{A}) \mid \varphi \, ; \, \varphi \mid \varphi\mathtt{+} \mid \varphi \ \mathtt{OR} \ \varphi.$$

Here, R ranges over relation names, A over variables in $\mathbf{L}$ and $P(\bar{A})$ over atoms over $\mathcal{P}$.

In Section 8.1 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 (:), contiguous iteration ($\oplus$) and the strict selection strategy `STRICT` are not expressible in CEL*. Then, in Section 8.2, we study the expressiveness of CEL* extended with different contiguous operators from $\{ : , \oplus, \mathtt{STRICT}\}$.

8.1 Unary CEL and the $*$ property

We first observe the following property of CEL* formulas φ, whose proof is by induction on φ.

Lemma 8.2. *For every CEL* formula φ, if $\mu \in [\![\varphi]\!](S,i,j)$, then*

(1) $j \in \sup(\mu)$, *and*
(2) $\emptyset \neq \sup(\mu) \subseteq \{i,\ldots,j\}$.

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 φ.

Lemma 8.3. *For every CEL* formula φ, every stream S, all positions $i \leq j$, and all valuations μ it holds that, if $\mu \in [\![\varphi]\!](S,i,j)$ then also*

(1) $\mu \in [\![\varphi]\!](S, \min(\sup(\mu)), \max(\sup(\mu)))$; *and*
(2) $\mu \in [\![\varphi]\!](S,i',j)$, *for every $i' \leq i$.*

The $*$-property. We next introduce the $*$-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 $S[C\rangle$ refer to the subsequence of S induced by C, i.e.,

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

where $C = \{i_1,\ldots,i_k\}$ with $i_1 < i_2 < \cdots < i_k$. A *stream-function* is a function $f : \text{streams}(\mathcal{R}) \times \mathbb{N} \to 2^{\mathbf{C}}$, where streams($\mathcal{R}$) is the set of all $\mathcal{R}$-streams and $\mathbf{C}$ 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 φ (i.e. $f(S,n) = [\![\varphi]\!]_n(S)$) or by a CEA $\mathcal{A}$ (i.e. $f(S,n) = [\![\mathcal{A}]\!]_n(S)$). 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 $(S_1,C_1) =_* (S_2,C_2)$, if $S_1[C_1\rangle = S_2[C_2\rangle$.

Definition 8.4. 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 $(S,C) =_* (S',C')$.

A way to understand the $*$-property is to see S' as the result of fixing the tuples in S that are part of $S[C\rangle$ 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 $(S,C) =_* (S',C')$ 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 8.5. *The function that maps $(S,n) \mapsto [\![\varphi]\!]_n(S)$ has the $*$-property, for every (not necessarily unary) CEL* formula φ.*

Proof. Let φ be a CEL formula, let S be a stream, let n be a position and C a complex event. Lemma 8.2 already ensures that if $C \in [\![\varphi]\!]_n(S)$ then $n \in C$. It hence suffices to show that

$$\text{if } C \in [\![\varphi]\!]_n(S) \text{ and } (S, C) =_* (S', C') \text{ then also } C' \in [\![\varphi]\!]_{\max(C')}(S'). \tag{2}$$

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

$$\text{if } \mu \in [\!|\varphi|\!](S_1, i, j) \text{ and } S_1 =_\alpha S_2 \text{ then also } \alpha(\mu) \in [\!|\varphi|\!](S_2, \alpha(i), \alpha(j)). \tag{3}$$

From this (2) follows. Indeed, assume that $(S, C) =_* (S', C')$. If $C \in [\![\varphi]\!]_n(S)$ then there exists a valuation μ such that $C = \sup(\mu)$ and $\mu \in [\!|\varphi|\!](S, 0, n)$. By Lemma 8.2, C is non-empty, $C \subseteq \{0, \ldots, n\}$, and $n \in C$. In particular, $n = \max(C)$. Moreover, by Lemma 8.3, also $C \in [\![\varphi]\!](S, \min(C), n)$. 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 α be the correspondence that maps $i_m \mapsto i'_m$ for all $m \in [1, k]$. Then, because $S_1[C_1\rangle = S_2[C_2\rangle$ clearly, $S_1 =_\alpha S_2$ and $\alpha(C) = C'$. Moreover, because α is strictly increasing, $\max(C') = \max(\alpha(C)) = \alpha(\max(C)) = \alpha(n)$. Because $\mu \in [\![\varphi]\!](S, \min(C), \max(C))$ we know by (3) that $\alpha(\mu) \in [\!|\varphi|\!](S_2, \alpha(\min(C)), \alpha(\max(C)))$ and therefore $C' = \sup(\alpha(\mu)) \in [\![\varphi]\!](S_2, \alpha(\min(C)), \alpha(\max(C)))$. Then, by Lemma 8.3, also $C' \in [\![\varphi]\!](S_2, 0, \alpha(\max(C)))$. Hence, $C' \in [\![\varphi]\!]_{\max(C')}(S_2)$, as desired.

The proof of property (3) is by induction on φ. We only show the following illustrative case. Fix $S_1, S_2, i, j, \mu, \alpha$ and assume that $\{i, j\} \cup \sup(\mu) \subseteq \mathrm{dom}(\alpha)$, $\mu \in [\!|\varphi|\!](S_1, i, j)$, and $S_1 =_\alpha S_2$.

Case $\varphi = \phi_1 : \phi_2$. Then there exists $k \in \mathbb{N}$ and valuations μ_1 and μ_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 8.2, $k \in \sup(\mu_1) \subseteq \sup(\mu)$. It follows that $\{i, k\} \cup \sup(\mu_1) \subseteq \{i, j\} \cup \sup(\mu) \subseteq \mathrm{dom}(\alpha)$ and we may hence apply the induction hypothesis on ϕ_1 to obtain that $\alpha(\mu_1) \in [\!|\phi_1|\!](S_2, \alpha(i), \alpha(k))$. To apply the induction hypothesis on ϕ_2 we need a bit more work. First, from $\mu_2 \in [\!|\phi_2|\!](S_1, k+1, j)$ and Lemma 8.3 we obtain that also $\mu_2 \in [\!|\phi_2|\!](S_1, \min(\sup(\mu_2)), j)$. Clearly, $\sup(\mu_2) \subseteq \sup(\mu) \subseteq \mathrm{dom}(\alpha)$. It follows that $\{\min(\sup(\mu_2)), j\} \cup \sup(\mu_2) \subseteq \mathrm{dom}(\alpha)$ and we may hence apply the induction hypothesis on ϕ_2 to obtain that $\alpha(\mu_2) \in [\!|\phi_2|\!](S_2, \alpha(\min(\sup(\mu_2))), \alpha(j))$. 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 $\alpha(\mu_2) \in [\!|\phi_2|\!](S_2, \alpha(\min(\sup(\mu_2))), \alpha(j))$ using Lemma 8.3, provided that $\alpha(k)+1 \leq \alpha(\min(\sup(\mu_2)))$. To see that $\alpha(k)+1 \leq \alpha(\min(\sup(\mu_2)))$, we reason as follows. First, observe that because $\mu_2 \in [\!|\phi_2|\!](S_1, k+1, j)$ we know from Lemma 8.2 that $\sup(\mu_2) \subseteq \{k+1, \ldots, j\}$; therefore $k+1 \leq \min(\sup(\mu_2))$. Hence, $k < k+1 \leq \min(\sup(\mu_2))$. Then, because α is strictly increasing, we know that $\alpha(k) < \alpha(\min(\sup(\mu_2)))$. Therefore, $\alpha(k)+1 \leq \alpha(\min(\sup(\mu_2)))$. □

We note that the CEL operators in $\{:, \oplus, \pi_L,$ START, UNLESS$\}$, as well as the selection operators 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 UNLESS, consider $\varphi = R$ 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 $S_1 = t_0 t_1 \ldots$ 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, $\{1\} \in [\![\varphi]\!]_1(S_1)$, $\{1\} \notin [\![\varphi]\!]_1(S_2)$ and yet $(S_1, \{1\}) =_* (S_2, \{1\})$. Therefore, φ does not have the $*$-property, and is hence not expressible in unary CEL*. Similar patterns can be constructed for the other operators.

Corollary 8.6. *The CEL operators in* $\{:, \oplus, \pi_L,$ `START`, `UNLESS`$\}$, *as well as the selection operators* $\{$`NXT`, `STRICT`, `LAST`, `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 6.2). 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 8.7. Let $\mathcal{A} = (Q, \Delta, I, F)$ be a complex event automaton and $S = t_1 t_2 \ldots$ be a stream. A $*$-*run* of $\mathcal{A}$ over S ending at n is a sequence of transitions: $\rho^* : (q_0, 0) \xrightarrow{P_1/L_1} (q_1, i_1) \xrightarrow{P_2/L_2} \cdots \xrightarrow{P_k/L_k} (q_k, i_k)$ 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 ρ^* is an accepting $*$-run if $q_k \in F$. Furthermore, we denote the valuation induced by ρ^* as μ_{ρ^*} such that for every variable A, $\mu_{\rho^*}(A) = \{i_k \mid A \in L_i\}$. The set of all valuations generated by $\mathcal{A}$ over S under the $*$-semantics is defined as: $[\![\mathcal{A}]\!]^*_n(S) = \{\mu_{\rho^*} \mid \rho^*$ is an accepting $*$-run of $\mathcal{A}$ over S ending at $n\}$. The set of all complex events generated by $\mathcal{A}$ over S under the $*$-semantics is then $[\![\mathcal{A}]\!]_n(S) = \{\sup(\mu) \mid \mu \in [\![\mathcal{A}]\!]^*_n(S)\}$.

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 8.8. *If the stream-function defined by a CEA* $\mathcal{A}$ *has the* $*$*-property, then* $[\![\mathcal{A}]\!]_n(S) = [\![\mathcal{A}]\!]^*_n(S)$ *for every* S *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 ρ 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, $(S, C) =_* (S', C')$ 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 $S'[C'\rangle = S[C\rangle$ 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 σ of $\mathcal{A}$ over S exists. Because $S[C\rangle = S'[C'\rangle$, the $*$-run σ' of $\mathcal{A}$ over S' also exists, which must coincide with a normal run ρ' of $\mathcal{A}$ over S'. Because $\mathcal{A}$ defines a function with $*$-property, the accepting run ρ of $\mathcal{A}$ over S has to exist. □

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

Theorem 8.9. *At a complex event level, unary CEL* has the same expressive power as CEA under the $*$-semantics. Namely, for every unary CEL* formula φ 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 φ be a unary CEL* formula. By Theorem 6.2 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 8.8.

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* FALSE as a formula that is never satisfied. One way to define it is FALSE $= (R$ FILTER $\emptyset)$.

Like the proof of Theorem 6.3, 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 φ_{ij} that represents the complex events defined by the $*$-runs from q_i to q_j. To aid in the definition of φ_{ij} we will first define, for every $1 \leq k \leq |Q|$ the formula φ^k_{ij} 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 $\varphi^{|Q|}_{ij} = \varphi_{ij}$.

We define φ^k_{ij} 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 $\varphi^0_{ij} =$ FALSE; otherwise we define it as:

$$\varphi^0_{ij} = \rho_{P_1} \text{ OR } \rho_{P_2} \text{ OR } \ldots \text{ OR } \rho_{P_k}$$

where $P_1, \ldots, P_k$ are all the predicates of the marking transitions from q_i to q_j. Moreover, ρ_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 $\rho_P := (R_1 \text{ OR } \cdots \text{ OR } R_r)$ FILTER P' where P' is the second order extension of CEA predicate P. Next, the recursion is defined as:

$$\varphi^k_{ij} \;=\; \varphi^{k-1}_{ij} \text{ OR } (\varphi^{k-1}_{ik} \,;\, \varphi^{k-1}_{kj}) \text{ OR } (\varphi^{k-1}_{ik} \,;\, \varphi^{k-1}_{kk}{+} \,;\, \varphi^{k-1}_{kj})$$

Finally, the final formula $\varphi_\mathcal{A}$ is the result of considering φ_{1n}. The correctness of the construction can be proved by doing induction over the number of states. □

Notice that Theorem 8.9 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 $\mathcal{A} = (\{p, q\}, \{(p, (\text{tuples}(R), T), q)\}, \{p\}, \{q\})$ with $T \notin \mathcal{R}$, which for all n and S defines $[\![\mathcal{A}]\!]^*_n(S) = \{\mu \mid \mu(T) = \{n\} \wedge S[n] \in \text{tuples}(R)\}$. We claim that there is no unary CEL* formula φ such that $[\![\varphi]\!]_n(S) = [\![\mathcal{A}]\!]^*_n(S)$. The closest we can get is the formula $\varphi = R$ IN T, which recognizes

$[\![\varphi]\!]_n(S) = \{\mu \mid \mu(T) = \mu(R) = n \wedge S[n] \in \text{tuples}(R)\}$. Note that the valuations of $\mathcal{A}$ do not define a value for R, while the ones of φ do.

By combining Theorem 8.9 and Proposition 8.8 we get the following result.

Corollary 8.10. *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 φ such that $f(S,n) = [\![\varphi]\!]_n(S)$, for every S and n.*

We can now use this characterization to prove that AND and ALL are expressible in unary CEL*.

Corollary 8.11. *For every expression φ of the form φ_1 OP φ_2, with OP $\in \{\text{AND}, \text{ALL}\}$ and φ_1, φ_2 unary CEL* formulae, there is a unary CEL* formula φ' such that $[\![\varphi]\!]_n(S) = [\![\varphi']\!]_n(S)$ for every S and n.*

Proof. We illustrate the proof idea for $\varphi = \varphi_1$ AND φ_2. The idea for $\varphi = \varphi_1$ ALL φ_2 is similar.

The proof is by application of Corollary 8.10, 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 8.5, namely by showing the stronger property (3) introduced in that proof and using the fact that this stronger property is already proved for φ_1 and φ_2 in the Proof of Proposition 8.5. That AND can be defined by means of a CEA follows from Theorem 6.2. □

8.2 Strict Sequencing versus Strict Selection

There exists a relation between the 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 STRICT(R ; S) and 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 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.

We use the following notation to refer to extension of CEL*. Given a set of operators $O \subseteq \{\text{STRICT}, :, \oplus\}$, let CEL$^\star \cup O$ 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. For instance, STRICT(R ; T)+ is in CEL$^\star \cup \{\text{STRICT}\}$, but $(R : T)$+ is not. Moreover, to compare the expressiveness of two fragments with operators O and O', we say that fragment CEL$^\star \cup O$ is *contained* in fragment CEL$^\star \cup O'$, and use notation CEL$^\star \cup O \subseteqq$ CEL* O', to state that for every formula in CEL$^\star \cup O$ there exists a complex-event equivalent formula in CEL$^\star \cup O'$. We use CEL$^\star \cup O \subset$ CEL* O' to denote that CEL$^\star \cup O \subseteqq$ CEL* O' and CEL$^\star \cup O'$ is strictly more expressive: there exists a formula in CEL$^\star \cup O'$ for which no complex-event equivalent formula exists in CEL$^\star \cup O$.

Lastly, to compare two fragments we often add the restriction that both use only certain predicates. Given a fragment CEL$^\star \cup O$ and a set of predicates $\mathcal{P}$, we denote by CEL$^\star \cup O(\mathcal{P})$ the set of formulas in CEL$^\star \cup O$ that only use predicates of $\mathcal{P}$ in their filters. The expressiveness relation between fragments with different contiguous operators is summarized in Fig. 6.

As mentioned earlier, the formula $R : T$ in CEL$^\star \cup \{:\}$ has an equivalent formula STRICT(R ; T) in CEL$^\star \cup \{\text{STRICT}\}$. In a similar way, a more complex formula like R ; T+ : U also has an equivalent formula, namely: R ; $\big($STRICT(T ; U) OR T+ STRICT(T ; U)$\big)$. This kind of transformation can be extended to all formulas in CEL$^\star \cup \{ : \}$, as we show in Appendix B.

The expressive containment is strict: in CEL$^\star \cup \{\text{STRICT}\}$ the combination STRICT along with + allows to verify an unbounded number of strict-concatenation of events in the complex event,

$$\begin{array}{ccccc} \mathrm{CEL}^{\star} \cup \{:\}(\mathcal{P}) & \overset{(a)}{\subset} & & & \\ & & \mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{P}) & \overset{(c)}{\subseteq} & \mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{P}) \\ \mathrm{CEL}^{\star} \cup \{\oplus\}(\mathcal{P}) & \overset{(b)}{\not\subseteq} & & & \end{array}$$

Fig. 6. Expressiveness comparison between fragments with different contiguous operators. By Proposition 8.12, containment (a) holds and is strict for any set of predicates $\mathcal{P}$. By proposition 8.13, (c) holds for any set of predicates $\mathcal{P}$. By Theorem 8.14, (b) holds whenever $\mathcal{P}$ contains a binary equality predicate in which case (c) becomes strict. By Proposition 8.15, (c) becomes $\equiv$ whenever $\mathcal{P}$ is a set of unary predicates.

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

Proposition 8.12. *For any set of CEL predicates $\mathcal{P}$, $\mathrm{CEL}^{\star} \cup \{:\}(\mathcal{P}) \subset \mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{P})$.*

Conversely, operators : and $\oplus$ combined are in general more expressive than STRICT.

Proposition 8.13. *For any set of CEL predicates $\mathcal{P}$, $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{P}) \subseteq \mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{P})$.*

The proof comes from the idea that, given any formula φ in $\mathrm{CEL}^{\star}$, the formula φ' resulting after replacing in φ each ; and + by : and $\oplus$, respectively, will be equivalent to $\mathtt{STRICT}(\varphi)$, since both capture complex events satisfying pattern φ that form an interval.

The most interesting result of this section is that fragments $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}$ and $\mathrm{CEL}^{\star} \cup \{\oplus\}$ are incomparable. Clearly, $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\} \not\subseteq \mathrm{CEL}^{\star} \cup \{\oplus\}$ since formula $\mathtt{STRICT}(R\,;\,T)$ has no equivalent in $\mathrm{CEL}^{\star} \cup \{\oplus\}$. 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 $\mathrm{CEL}^{\star} \cup \{\oplus\} \not\subseteq \mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}$.

Theorem 8.14. *There exists a set $\mathcal{P}$ containing a binary CEL predicate and a formula φ in $\mathrm{CEL}^{\star} \cup \{\oplus\}(\mathcal{P})$ that is not equivalent to any formula in $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{P})$.*

From this result, it follows that the containment $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{P}) \subset \mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{P})$ 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, STRICT can be shown to be just as expressive as : and $\oplus$ together.

Proposition 8.15. *For a set of unary CEL predicates $\mathcal{U}$, $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{U}) \equiv \mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{U})$, that is, $\mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{U}) \subseteq \mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{U})$ and $\mathrm{CEL}^{\star} \cup \{:, \oplus\}(\mathcal{U}) \subseteq \mathrm{CEL}^{\star} \cup \{\mathtt{STRICT}\}(\mathcal{U})$.*

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 STRICT, at least in terms of expressiveness. However, the decision depends mostly on the nature of the predicates that are going to be supported.

9 ALGORITHMS FOR EVALUATING CEA

In this section we show how to efficiently evaluate CEA. We start by formalizing the notion of *efficient evaluation* that we consider, and then describe a CEA evaluation algorithm that satisfies these efficiency requirements.

9.1 Efficiency in Complex Event Recognition

Given a unary CEL formula φ and a stream $S = t_0 t_1 \ldots$ we want to evaluate φ over S by reading the events of S in order and, after each new read event t_n, computing $[\![\varphi]\!]_n(S)$. Because φ can be compiled into a CEA $\mathcal{A}_\varphi$ (cf. Section 6), we can reduce this problem to computing $[\![\mathcal{A}_\varphi]\!]_n(S)$. This hence yields the following enumeration problem.

Problem:	EvaluationCEA
Input:	A CEA $\mathcal{A}$ and a stream $S = t_0 t_1 \ldots$
Output:	Enumerate the set $[\![\mathcal{A}]\!]_n(S)$ after reading t_n, for every $n \geq 0$

We next specify what sort of algorithms we allow to solve the above problem, and what the efficiency guarantees are that we would like to achieve. For the algorithm, we assume the model of Random Access Machines (RAM) with uniform cost measure, and addition and subtraction as basic operations [8]. 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 [36, 59]. Further, we assume that the stream S can be read by calling a special instruction $\texttt{yield}_S$ 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 [40, 47] 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 $\mathcal{A}$ be a CEA and let $\mathcal{U}$ be the set of CEA predicates used by transitions of $\mathcal{A}$. Furthermore, if t is a tuple, then let $|t|$ denote the number of RAM registers required to encode t. Let $W_\mathcal{U} : \mathbb{N} \to \mathbb{N}$ be a function such that, for any tuple t and any $P \in \mathcal{U}$, the time needed for verifying whether $t \in P$ is bounded by $W_\mathcal{U}(|t|)$ (i.e., $W_\mathcal{U}(|t|)$ is the worst-case time of evaluating any predicate in $\mathcal{U}$). For a function $f : \mathbb{N} \to \mathbb{N}$, a *CER evaluation algorithm* with f-update time and output-linear delay is an algorithm that evaluates a CEA $\mathcal{A}$ over a stream $S = t_0 t_1 \ldots$. It receives as input $\mathcal{A}$ and reads S by calling the $\texttt{yield}_S$ 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 $O(f(|\mathcal{A}|) \cdot W_\mathcal{U}(|t_n|))$.

(II) The second phase, called the *enumeration phase*, occurs immediately after the first phase and outputs $[\![\mathcal{A}]\!]_n(S)$ 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 $[\![\mathcal{A}]\!]_n(S)$, (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.[2] 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 $\mathcal{A}$ on S. Let $\text{time}_i(\mathcal{A}, S, n)$ denote the time, during the enumeration phase of the n-th event $\mathcal{A}$ on S, that the algorithm writes the i-th #, for $1 \leq i \leq m_n$. Define $\text{delay}_i(\mathcal{A}, S, n) = \text{time}_{i+1}(\mathcal{A}, S, n) - \text{time}_i(\mathcal{A}, S, n)$ for $1 \leq i \leq m_n - 1$. Then we say that the algorithm has *output-linear delay* if there exists a constant k such that for all stream positions n, (1) if $[\![\mathcal{A}]\!]_n(S)$ is non-empty then $\text{delay}_i(\mathcal{A}, S, n) \leq k \cdot |C_i|$ for every every $i \leq |[\![\mathcal{A}]\!]_n(S)|$; and (2) if $[\![\mathcal{A}]\!]_n(S) = 0$ then $\text{delay}_i(\mathcal{A}, S, 1) = k$. In other words, the time between writing the i-th # and $i+1$-th # is at most linear in the size of i-th output C_i of $[\![\mathcal{A}]\!]_n(S)$, and is constant if $[\![\mathcal{A}]\!]_n(S)$ is empty.

We remark that (I) is a natural restriction imposed in the streaming literature [47], while (II) is the minimum requirement if an arbitrarily large set of arbitrarily large outputs must be produced [60]. Concerning (I), note that the update time $O(f(|\mathcal{A}|) \cdot W_{\mathcal{U}}(|t_n|))$ is linear in $W_{\mathcal{U}}(|t_n|)$, the worst-case time of evaluating any predicate in $\mathcal{U}$ over t_n. This bound depends on the predicates provided by the CER engine. Indeed, if t_n is of constant size (which is the case, for example, when the schema is fixed and all data types are integers), then $W_{\mathcal{U}}(|t_n|)$ takes constant time. Furthermore, in practice the CEA $\mathcal{A}$ is generally small, especially compared to the unbounded length of the stream. Viewed from the prespective of data complexity [4], an update time of $O(f(|\mathcal{A}|) \cdot W_{\mathcal{U}}(|t_n|))$ then amounts to constant update time. In conclusion, under the assumption of a constant schema, constant CEA, and all data values fitting in a single register, (I) implies that the update phase takes constant time update per tuple.

9.2 An evaluation algorithm for CEA with linear update time

Having a good notion of efficiency, we proceed to show how to solve EvaluationCEA with the above guarantees. Specifically, in this section we show the following result.

Theorem 9.1. *For every CEA $\mathcal{A}$ in normal form, there is a CER evaluation algorithm with $|\mathcal{A}|$-update time and output-linear delay.*

Here, CEA $\mathcal{A}$ is in *normal form* if (1) it is *single-variable*: there exists $A \in \mathbf{L}$ 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 5.5 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 9.2. *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 $A \in \mathbf{L}$. 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 ρ' of $\mathcal{A}'$ will mark an event by A if, and only if, a corresponding run ρ 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. □

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[3], the algorithm of Theorem 9.1

[2] Actually, it will be possible to run the enumeration phase separately, without interrupting the update phase (see Section 9.2).
[3] Note that the I/O-determinizing a single-variable CEA using the method of Proposition 5.5 will result again in a single-variable CEA.

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 9.3. *For every CEA $\mathcal{A}$, there is a CER evaluation algorithm with $2^{|\mathcal{A}|}$-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 7.2.

The rest of this section is devoted to proving Theorem 9.1. 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\perp$, and every path to $\perp$ 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 $G_{\mathcal{D}}$ that is depicted at the right of Fig 7: every complex event $C \in C$ corresponds to a path from node 5 or 6 in $G_{\mathcal{D}}$ to $\perp$. Note that all nodes in $G_{\mathcal{D}}$, 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. 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 C from $G_{\mathcal{D}}$ without repetitions simply by enumerating all paths from nodes 5 and 6 to $\perp$ in $G_{\mathcal{D}}$. 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 $O(|\mathcal{A}| \cdot W_{\mathcal{U}}(|t|))$ 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

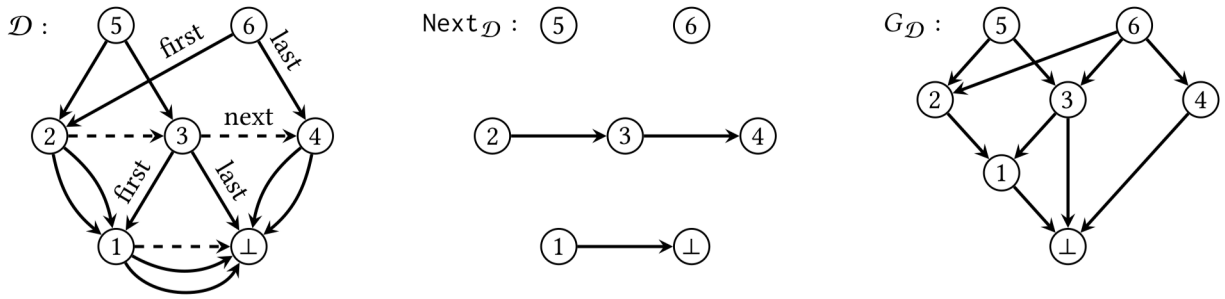

Fig. 7. An example of a Compact Complex Event Set. On the left, we show its structure $\mathcal{D}$ where nodes are labeled with its corresponding event, and the `first`, `last`, and `next` functions are represented as edges (see node 3 and 6). The directed graphs on the middle and right of the figure are the $\texttt{Next}_{\mathcal{D}}$ and $G_{\mathcal{D}}$ of $\mathcal{D}$, respectively.

this way of the DAG $G_{\mathcal{D}}$ of Fig. 7 (right) is shown as the DAG $\mathcal{D}$ in Fig. 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 *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:

$$\mathcal{D} \;=\; (\mathsf{N}, \bot, \texttt{event}, \texttt{first}, \texttt{last}, \texttt{next}) \qquad (*)$$

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

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

Let $\texttt{Next}_{\mathcal{D}}$ be the subgraph of $\mathcal{D}$ formed by only the `next` edges, i.e., $\texttt{Next}_{\mathcal{D}} = (\mathsf{N}, \{(\mathsf{n}, \texttt{next}(\mathsf{n})) \mid \mathsf{n} \in \mathsf{N} \wedge \texttt{next}(\mathsf{n}) \neq \emptyset\})$. The first two restrictions that $\mathcal{D}$ must satisfy are:

(1) The graph $\texttt{Next}_{\mathcal{D}}$ is acyclic.
(2) For every $\mathsf{n} \in \mathsf{N}$, there is a path from $\texttt{first}(\mathsf{n})$ to $\texttt{last}(\mathsf{n})$ in $\texttt{Next}_{\mathcal{D}}$.

Given that `next` is a partial function, (1) implies that $\texttt{Next}_{\mathcal{D}}$ consists of a set of paths. Furthermore, (2) implies that, for each n, the pair $(\texttt{first}(\mathsf{n}), \texttt{last}(\mathsf{n}))$ represents a list of nodes, starting in $\texttt{first}(\mathsf{n})$ and ending in $\texttt{last}(\mathsf{n})$. In the middle of Fig. 7, we display the graph $\texttt{Next}_{\mathcal{D}}$ for our example. The reader can check that the aforementioned properties are satisfied.

We say that $\mathsf{n} \in \mathsf{N}$ *can reach* $\mathsf{n}' \in \mathsf{N}$ if, and only if, there is a (possibly empty) directed path from n to n' in $\texttt{Next}_{\mathcal{D}}$. If this is the case, we define $\texttt{list}(\mathsf{n}, \mathsf{n}') = \mathsf{n}_0, \ldots, \mathsf{n}_k$ such that $\mathsf{n}_0 = \mathsf{n}$, $\mathsf{n}_{i+1} = \texttt{next}(\mathsf{n}_i)$, and $\mathsf{n}_k = \mathsf{n}'$. By Property (2) we know that, for every $\mathsf{n} \in \mathsf{N}$, $\texttt{first}(\mathsf{n})$ can reach $\texttt{last}(\mathsf{n})$ and, moreover, $\texttt{list}(\texttt{first}(\mathsf{n}), \texttt{last}(\mathsf{n}))$ is the list of nodes between them. By some abuse of notation, we will write $\mathsf{n}'' \in \texttt{list}(\mathsf{n}, \mathsf{n}')$ to denote that n'' appears in $\texttt{list}(\mathsf{n}, \mathsf{n}')$.

For the third restriction, define the directed graph:

$$G_{\mathcal{D}} \;=\; (\mathsf{N}, \{(\mathsf{n}, \mathsf{n}') \mid \mathsf{n}' \in \texttt{list}(\texttt{first}(\mathsf{n}), \texttt{last}(\mathsf{n}))\}).$$

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

(3) The graph $G_{\mathcal{D}}$ is acyclic.

In fact, as already mentioned, the purpose of $\mathcal{D}$ is to represent the acyclic graph $G_{\mathcal{D}}$. Note that $\bot$ is the only node without out-edges in $\mathcal{D}$ (i.e., `first` and `last` is not defined for $\bot$) and, hence, by the acyclicity of $G_{\mathcal{D}}$, every path of $G_{\mathcal{D}}$ ends in $\bot$. For an illustration, in Fig. 7 (right) we provide the acyclic graph $G_{\mathcal{D}}$ for our example.

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 $\bigcup_{C' \in \mathsf{CE}(\mathsf{n}_1,\mathsf{n}_2)} \{C \cup C'\}$.

```
1: procedure ENUMERATE(D, n1, n2, C)
2:     if n1 = ⊥ then
3:         Output(C)
4:     else
5:         ENUMERATE(D, first(n1), last(n1), C ∪ {event(n1)})
6:     if n1 ≠ n2 then
7:         ENUMERATE(D, next(n1), n2, C)
```

Until now we have not used the event function of CCES $\mathcal{D}$, which is used to represent complex events. Let $\pi = \mathsf{n}_0, \ldots, \mathsf{n}_k$ be a path in $G_\mathcal{D}$ ending in $\bot$ (i.e., $\mathsf{n}_k = \bot$). We define the complex event associated to π as $\mathsf{CE}(\pi) = \{\mathsf{event}(\mathsf{n}_0), \ldots, \mathsf{event}(\mathsf{n}_{k-1})\}$, and the set of complex events defined by $\mathsf{n} \in \mathsf{N}$ as $\mathsf{CE}(\mathsf{n}) = \{\mathsf{CE}(\pi) \mid \pi \text{ is a path in } G_\mathcal{D} \text{ starting in } \mathsf{n} \text{ and ending in } \bot\}$. Furthermore, for $\mathsf{n}, \mathsf{n}' \in \mathsf{N}$ such that n can reach n' we define $\mathsf{CE}(\mathsf{n}, \mathsf{n}') = \bigcup_{\mathsf{n}'' \in \mathsf{list}(\mathsf{n},\mathsf{n}')} \mathsf{CE}(\mathsf{n}'')$. In our running example, we can check that $\mathsf{CE}(5) = \{\{1, 2, 5\}, \{1, 3, 5\}, \{3, 5\}\}$, which are all paths from 5 to $\bot$ (see Fig. 7, right). One can also check that, for example, $\mathsf{CE}(2, 4) = \{\{1, 2\}, \{1, 3\}, \{3\}, \{4\}\}$.

While the graph $G_\mathcal{D}$ 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 $\mathsf{n} \in \mathsf{N} \setminus \{\bot\}$ and $\mathsf{n}' \in \mathsf{list}(\mathsf{first}(\mathsf{n}), \mathsf{last}(\mathsf{n}))$, if $\mathsf{n}' \neq \bot$, then $\mathsf{event}(\mathsf{n}) > \mathsf{event}(\mathsf{n}')$.
(5) For every $\mathsf{n}, \mathsf{n}' \in \mathsf{N}$ such that n can reach n', and for every pair of different paths π and π' starting in some node in $\mathsf{list}(\mathsf{n}, \mathsf{n}')$ and ending in $\bot$, it holds that $\mathsf{CE}(\pi) \neq \mathsf{CE}(\pi')$.

Property (4) forces that the (positions of the) events of a complex event represented by a path π in $G_\mathcal{D}$ are in decreasing order and, therefore, positions cannot be repeated in a path. Property (5) enforces that there are no "repetitions" in the set $\mathsf{CE}(\mathsf{n}, \mathsf{n}')$, namely, two paths that define the same complex event. This fact will allow us to enumerate the complex events in $\mathsf{CE}(\mathsf{n}, \mathsf{n}')$, 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. 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 $(\mathsf{n}_1, \mathsf{n}_2)$ encodes the set of complex events $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$.

To enumerate $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$ 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 $\bigcup_{C' \in \mathsf{CE}(\mathsf{n}_1,\mathsf{n}_2)} \{C \cup C'\}$. 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 $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$. As a special case, the initial call to ENUMERATE will set $C = \emptyset$ and the output will be $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$.

Recall the graph $G_\mathcal{D}$ represented by $\mathcal{D}$. Algorithm 1 enumerates $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$ by traversing $G_\mathcal{D}$ recursively. For the base case, when $\mathsf{n}_1 = \bot$, the algorithm reaches the sink node of $G_\mathcal{D}$ and it prints C. For the recursive case, it iterates with n over the list $\mathsf{list}(\mathsf{n}_1, \mathsf{n}_2)$ and enumerates $\mathsf{CE}(\mathsf{n}) = \{\{\mathsf{event}(\mathsf{n})\} \cup C' \mid C' \in \mathsf{CE}(\mathsf{first}(\mathsf{n}), \mathsf{last}(\mathsf{n}))\}$. In other words, ENUMERATE does a depth-first search of all paths of $G_\mathcal{D}$ that start from some node in $\mathsf{list}(\mathsf{n}_1, \mathsf{n}_2)$. The events of the path are stored in C, and each time that the sink node of $G_\mathcal{D}$ (i.e., $\bot$) 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 $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$ with output-linear delay, one has to notice two facts. First, recall that $\mathcal{D}$ satisfies Property (5), which means that $\mathtt{CE}(\pi) \neq \mathtt{CE}(\pi')$ for every pair of different paths π and π' starting in some node in $\mathtt{list}(\mathsf{n}_1, \mathsf{n}_2)$ and ending in $\bot$. Therefore, the enumeration of complex events gives no repetitions. Second, after C is printed by Enumerate (line 3), the recursion performs a "backtracking" until the next node that satisfies $\mathsf{n}_1 \neq \mathsf{n}_2$ (line 6). Then the recursion calls Enumerate going into $G_{\mathcal{D}}$ until $\bot$ 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 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 9.1) 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 $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$ with output-linear delay as expected.

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

$$\begin{array}{rcll} (\mathcal{D}, \bot) & := & \mathtt{init}() & \\ (\mathcal{D}', \mathsf{n}) & := & \mathtt{extend}(\mathcal{D}, i, \mathsf{n}_1, \mathsf{n}_2) & \text{s.t. } \mathsf{n}_1 \text{ can reach } \mathsf{n}_2 \\ \mathcal{D}' & := & \mathsf{append}(\mathcal{D}, \mathsf{n}_1, \mathsf{n}_2) & \text{s.t. } \mathtt{next}(\mathsf{n}_1) = \emptyset \end{array}$$

The first operation is to initialize a new CCES $\mathcal{D}$ with its corresponding special node $\bot$. The other two operations receive a CCES $\mathcal{D} = (\mathsf{N}, \bot, \mathtt{event}, \mathtt{first}, \mathtt{last}, \mathtt{next})$ and output an extension of $\mathcal{D}$, denoted as $\mathcal{D}' = (\mathsf{N}', \bot, \mathtt{event}', \mathtt{first}', \mathtt{last}', \mathtt{next}')$. 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 $\mathtt{event}'(\mathsf{n}) = i$, $\mathtt{first}'(\mathsf{n}) = \mathsf{n}_1$, $\mathtt{last}'(\mathsf{n}) = \mathsf{n}_2$, and $\mathtt{next}'(\mathsf{n})$ is not defined (i.e., $\mathtt{next}'(\mathsf{n}) = \emptyset$). In particular, it holds that $\mathsf{N}' = \mathsf{N} \cup \{\mathsf{n}\}$ and $f'(\mathsf{n}') = f(\mathsf{n}')$ for every $f \in \{\mathtt{event}, \mathtt{next}, \mathtt{first}, \mathtt{last}\}$ and $\mathsf{n}' \in \mathsf{N}$.

The append operation receives $\mathcal{D}$ and two nodes n_1 and n_2 of $\mathcal{D}$ such that $\mathtt{next}(\mathsf{n}_1) = \emptyset$. It outputs the extension $\mathcal{D}'$ of $\mathcal{D}$ such that $\mathtt{next}'(\mathsf{n}_1) = \mathsf{n}_2$. Similar to for extend, all the other components of $\mathcal{D}$ remain the same, namely, $\mathsf{N}' = \mathsf{N}$, $\mathtt{next}'(\mathsf{n}) = \mathtt{next}(\mathsf{n})$ for every $\mathsf{n} \neq \mathsf{n}_1$, and $f'(\mathsf{n}') = f(\mathsf{n}')$ for every $f \in \{\mathtt{event}, \mathtt{first}, \mathtt{last}\}$ and $\mathsf{n}' \in \mathsf{N}$. Intuitively, append will be used to concatenate one list with another. Recall that a pair $(\mathsf{n}_1, \mathsf{n}_2)$ such that n_1 can reach n_2 represents the list of nodes $\mathtt{list}(\mathsf{n}_1, \mathsf{n}_2)$. Then, if we have two pairs $(\mathsf{n}_1, \mathsf{n}_2)$ and $(\mathsf{n}'_1, \mathsf{n}'_2)$ such that $\mathtt{next}(\mathsf{n}_2) = \emptyset$, after applying $\mathsf{append}(\mathcal{D}, \mathsf{n}_2, \mathsf{n}'_1)$, n'_2 will be reachable from n_1 in $\mathcal{D}'$, and $(\mathsf{n}_1, \mathsf{n}'_2)$ will be the concatenation of $\mathtt{list}(\mathsf{n}_1, \mathsf{n}_2)$ and $\mathtt{list}(\mathsf{n}'_1, \mathsf{n}'_2)$.

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 $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$ represented by $(\mathsf{n}_1, \mathsf{n}_2)$ in $\mathcal{D}$ are preserved in $\mathcal{D}'$. More specifically, for any pair of nodes $(\mathsf{n}_1, \mathsf{n}_2)$ of $\mathcal{D}$ such that n_1 can reach n_2, it holds that $\textsc{Enumerate}(\mathcal{D}, \mathsf{n}_1, \mathsf{n}_2, C)$ and $\textsc{Enumerate}(\mathcal{D}', \mathsf{n}_1, \mathsf{n}_2, C)$ give the same output. In other words, CCES is a *partially persistent* data structure [35], 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.

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

```
1:  procedure EVALUATE(A, S)
2:      (D, ⊥) ← init()
3:      T ← ∅
4:      T[q_0] ← (⊥, ⊥)
5:      while t_i ← yield_S do
6:          T' ← ∅
7:          for all p ∈ dom(T) do
8:              (n_1, n_2) ← T[p]
9:              if q_• ← Δ(p, t_i, •) then
10:                 (D, n) ← extend(D, i, n_1, n_2)
11:                 (D, T') ← ADD(D, T', q_•, n, n)
12:             if q_∘ ← Δ(p, t_i, ∘) then
13:                 (D, T') ← ADD(D, T', q_∘, n_1, n_2)
14:         T ← T'
15:         OUTPUT(T, F)
16: procedure ADD(D, T, q, n_1, n_2)
17:     if T[q] is not defined then
18:         T[q] ← (n_1, n_2)
19:     else
20:         (n'_1, n'_2) ← T[q]
21:         D ← append(D, n'_2, n_1)
22:         T[q] ← (n'_1, n_2)
23:     return (D, T)
24:
25: procedure OUTPUT(T, F)
26:     for all p ∈ F ∩ dom(T) do
27:         (n_1, n_2) ← T[p]
28:         ENUMERATE(D, n_1, n_2, ∅)
```

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 $\mathcal{D}$ to store complex events for each active state. For each active state q we maintain a pair $(\mathsf{n}_1, \mathsf{n}_2)$ such that n_1 can reach n_2 and $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$ 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 extend and append over $\mathcal{D}$. After the update is done, we check for all active states that are final and output $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$, that is, by using the procedure ENUMERATE.

To remember the set of active states, we use a *lookup table* indexed by states that stores a pair $(\mathsf{n}_1, \mathsf{n}_2)$ for each state. Formally, this lookup table is a partial function $\mathrm{T} : Q \rightharpoonup \mathsf{N} \times \mathsf{N}$ for a set of nodes N. We write dom(T) to denote all the states that have an entry in T and $\emptyset$ to denote the empty lookup table. Then, if $q \in \mathrm{dom}(\mathrm{T})$, we write $\mathrm{T}[q]$ to retrieve the pair of nodes stored for q, and we say that $\mathrm{T}[q]$ is not defined if $q \notin \mathrm{dom}(\mathrm{T})$. We use the notation $\mathrm{T}[q] \leftarrow (\mathsf{n}_1, \mathsf{n}_2)$ to declare an update on the entry q of T with $(\mathsf{n}_1, \mathsf{n}_2)$. 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 $\mathcal{D}$ 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 dom(T)) and "firing" the $\bullet$- and $\circ$-transitions of p with t_i. To ease the notation here, we extend Δ 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 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

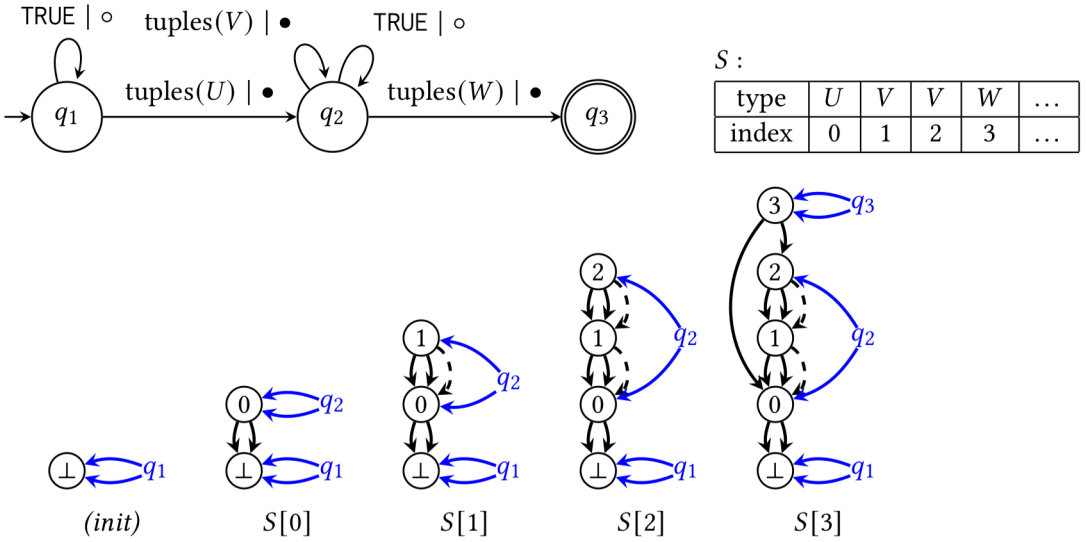

Fig. 8. A CEA $\mathcal{A}$ in normal form (top left) and stream S (top right) and an illustration of Algorithm 2 on $\mathcal{A}$ and S (bottom figures).

in $(\mathsf{n}_1, \mathsf{n}_2)$. If the $\bullet$-transition is fired, then we must extend each complex event represented by $(\mathsf{n}_1, \mathsf{n}_2)$ with the new event t_i (i.e., position i). For this, we use the `extend` method of $\mathcal{D}$ (line 10), by getting the new version of $\mathcal{D}$ 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 Add (defined at the right column of Algorithm 2). Similarly, if the $\circ$-transition is fired, then we add the list $(\mathsf{n}_1, \mathsf{n}_2)$ directly to the list of $q_\circ$ in T′ (i.e. no extension is needed).

The method $\textsc{Add}(\mathcal{D}, \mathrm{T}, q, \mathsf{n}_1, \mathsf{n}_2)$ is a common procedure to both transitions and is in charge of appending the list represented by $(\mathsf{n}_1, \mathsf{n}_2)$ to the list of q stored in T. For this, we first need to check whether there is a list in $\mathrm{T}[q]$ or not, and add it directly if not (lines 17-18). Otherwise, we retrieve $\mathrm{T}[q]$ in $(\mathsf{n}'_1, \mathsf{n}'_2)$, append $(\mathsf{n}_1, \mathsf{n}_2)$ to $(\mathsf{n}'_1, \mathsf{n}'_2)$, and store the new list $(\mathsf{n}'_1, \mathsf{n}_2)$ in $\mathrm{T}[q]$ (lines 20-22). Finally, we output the new version of $\mathcal{D}$ and the updated lookup table T.

After iterating over all states $p \in \mathrm{dom}(\mathrm{T})$, 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 $\textsc{Output}(T, F)$ (line 15). The Output procedure (lines 25-28) checks which of the active states are final. For such a state p, it instantiates its list $\mathrm{T}[p]$ in $(\mathsf{n}_1, \mathsf{n}_2)$ and call the enumeration procedure Enumerate with $(\mathsf{n}_1, \mathsf{n}_2)$, starting with an empty complex event.

An example. Consider the CEA $\mathcal{A}$ and stream S from Fig. 8. $\mathcal{A}$ is in normal form, and corresponds to the CEL formula $U\,;\,[(V{+}\,;\,W)\ \texttt{OR}\ W]$ (it is a simplified version of the CEA in Fig. 4). In particular,

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

The lower half of Fig. 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 $\mathcal{D}$ and T after initialization (lines 2–4) while the others depict $\mathcal{D}$ 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:

$(q_1, \mathsf{TRUE}, \circ, q_1)$ and $(q_1, \text{tuples}(U), \bullet, q_2)$. When $(q_1, \mathsf{TRUE}, \circ, q_1)$ 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 `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. 8.

When processing $S[1]$, there are three applicable transitions: $(q_1, \mathsf{TRUE}, \circ, q_1)$, $(q_2, \text{tuples}(V), \bullet, q_2)$, and $(q_2, \mathsf{TRUE}, \circ, q_2)$. When $(q_1, \mathsf{TRUE}, \circ, q_1)$ 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 `extend` to create node 1 with child list $(0, 0)$, and line 11 sets $T'[q_2] = (1, 1)$. When $(q_2, \mathsf{TRUE}, \circ, q_2)$ 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. 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 $\textsc{Enumerate}(\mathcal{D}, 3, 3, \emptyset)$ to be called. We invite the reader to check that this correctly enumerates $[\![\mathcal{A}]\!]_3(S)$.

Correctness. For an accepting run ρ of $\mathcal{A}$, let us write $\text{events}(\rho)$ for the complex event produced by ρ. 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 $\mathcal{D}$ 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 $p \in Q$, $p \in \text{dom}(\mathrm{T}_i)$ if, and only if, p is an active state of $\mathcal{A}$ after reading $t_0 \dots t_{i-1}$.

(‡) For every $p \in \text{dom}(\mathrm{T}_i)$ and $(\mathsf{n}_1, \mathsf{n}_2) = \mathrm{T}_i[p]$, it holds that $C \in \mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$ if, and only if, there exists a run ρ of $\mathcal{A}$ over $t_0 \dots t_{i-1}$ ending in p such that $\text{events}(\rho) = C$.

Properties (1) to (3) of the CCES $\mathcal{D}$ are satisfied given that $\mathcal{D}$ is accessed only through methods `extend` and `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 $\mathsf{next}(\mathsf{n}_2) = \emptyset$ for each $p \in \text{dom}(\mathrm{T})$ and $(\mathsf{n}_1, \mathsf{n}_2) = \mathrm{T}[p]$. This holds for the initial case $\mathrm{T}[q_0]$ and is preserved after each call to `extend` and `append`. Property (4) holds given that we always extend $\mathcal{D}$ 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 $t_0 \dots t_{i-1}$ 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 $A_i = \text{dom}(\mathrm{T}_i)$ 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 $[\![\mathcal{A}]\!]^q_i(S)$ as the set of all complex events C such that there exists a run ρ of $\mathcal{A}$ over $t_0 \dots t_{i-1}$ ending in q such that $\text{events}(\rho) = C$. Similar than for A_i, the set $[\![\mathcal{A}]\!]^q_i(S)$ can be recursively defined as follow. For the base case, we define $[\![\mathcal{A}]\!]^{q_0}_0(S) = \{\emptyset\}$ and $[\![\mathcal{A}]\!]^q_0(S) = \emptyset$ for every $q \neq q_0$. For any $i > 0$ and $q \in Q$, the set $[\![\mathcal{A}]\!]^q_i(S)$ can be defined as:

$$[\![\mathcal{A}]\!]^q_i(S) = \bigcup_{\substack{p \in A_{i-1} : \\ \Delta(p, t_i, \bullet) = q}} \left\{ C \cup \{i\} \mid C \in [\![\mathcal{A}]\!]^p_{i-1}(S) \right\} \quad \cup \quad \bigcup_{\substack{p \in A_{i-1} : \\ \Delta(p, t_i, \circ) = q}} [\![\mathcal{A}]\!]^p_{i-1}(S) \qquad (**)$$

To prove (‡) we need to show that $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2) = [\![\mathcal{A}]\!]^q_i(S)$ for every $q \in \text{dom}(\mathrm{T}_i)$ and $(\mathsf{n}_1, \mathsf{n}_2) = \mathrm{T}_i[q]$. The goal is to prove this equivalence by induction on i and using the recursive definition of $[\![\mathcal{A}]\!]^q_i(S)$. For the base case, Algorithm 2 initializes T_0 with $\mathrm{T}_0[q_0] = (\bot, \bot)$ and $\mathrm{T}_0[q] = \emptyset$ for all $q \neq q_0$ (lines 3-4). Here, we can check that $\mathsf{CE}(\bot, \bot) = \{\emptyset\} = [\![\mathcal{A}]\!]^{q_0}_0(S)$ and $[\![\mathcal{A}]\!]^q_0(S) = \emptyset$ for every $q \neq q_0$. For the inductive case, note that in lines 7-13 we iterate over each $p \in \text{dom}(\mathrm{T}_{i-1}) = A_{i-1}$ (by invariant (†)) and pick $(\mathsf{n}_1, \mathsf{n}_2) = \mathrm{T}_{i-1}[p]$. Moreover, $\mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2) = [\![\mathcal{A}]\!]^p_{i-1}(S)$ by inductive hypothesis. Then, if $q = \Delta(p, t_i, \bullet)$, we extend each $C \in \mathsf{CE}(\mathsf{n}_1, \mathsf{n}_2)$ with i and add this to the results in $\mathrm{T}_i[q]$. This is

equivalent to the left union of (**). Instead, if $q = \Delta(p, t_i, \circ)$, we add $\mathtt{CE}(\mathsf{n}_1, \mathsf{n}_2)$ directly to the results in $\mathrm{T}_i[q]$, which is equivalent to the right union of (**). In other words, after the for-loop of lines 7-13 is done, $\mathrm{T}_i[q] = (\mathsf{n}'_1, \mathsf{n}'_2)$ satisfies that $\mathtt{CE}(\mathsf{n}'_1, \mathsf{n}'_2) = [\![\mathcal{A}]\!]^q_i(S)$ by equation (**). This concludes the proof for invariant (‡).

To finish the correctness of Algorithm 2, we note that for each call to $\mathtt{yield}_S$ 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 $W_{\mathcal{U}}$. Furthermore, checking whether there is an output or not takes time proportional to $|Q|$. Overall, the update phase takes time proportional to $|\mathcal{A}| \cdot W_{\mathcal{U}}(|t_i|)$ as expected.

10 CONCLUSIONS

This paper settles new foundations for CER systems, stimulating new research directions. In particular, it formalizes the fundamental aspects of CER languages, leaving other features outside in order to keep the language and analysis simple. These features include aggregation, consumption policies, among others. The natural next step is to extend CEL gradually with these features to establish a more complete and formal 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 more fundamental direction has to do with the study of the inherent complexity of the evaluation problem, in order to know which extensions of CEL can be evaluated with strong guarantees like output-linear delay enumeration. We believe that understanding the relation between expressivity and evaluation complexity is fundamental for the design of CER languages and the implementation of CER systems.

ACKNOWLEDGMENTS

A. Grez, C. Riveros, and M. Ugarte were supported by ANID - Millennium Science Initiative Program - Code ICN17_002. S. Vansummeren was supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University under Grant No. BOF20ZAP02.

REFERENCES

[1] [n.d.]. Apache FlinkCEP. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/libs/cep/. [Online; accessed 14-June-2021.

[2] [n.d.]. Esper Enterprise Edition website. http://www.espertech.com/. Accessed on 2021-01-28.

[3] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: A Data Stream Management System. In *SIGMOD*.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley.

[5] Asaf Adi and Opher Etzion. 2004. Amit-the situation manager. *VLDB Journal* (2004).

[6] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *SIGMOD*.

[7] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. North Holland, 255–300.

[8] Alfred V Aho and John E Hopcroft. 1974. *The design and analysis of computer algorithms.* Pearson Education India.

[9] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. 2008. Plan-based complex event detection across distributed sources. *VLDB* (2008).

[10] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. 2017. Probabilistic Complex Event Recognition: A Survey. *ACM Comput. Surv.* 50, 5 (2017), 71:1–71:31. https://doi.org/10.1145/3117809

[11] Rajeev Alur, Loris DAntoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. 2013. Regular Functions and Cost Register Automata. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. 13–22. https:

//doi.org/10.1109/LICS.2013.65
[12] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. 2020. Streamable regular transductions. *Theoretical Computer Science* 807 (2020), 15–41. https://doi.org/10.1016/j.tcs.2019.11.018 In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part II.
[13] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. 2010. A rule-based language for complex event processing and reasoning. In *RR*.
[14] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *SIGMOD*.
[15] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* (2006).
[16] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex Event Recognition Languages: Tutorial. In *DEBS*. ACM, 7–10.
[17] Alexander Artikis, Marek Sergot, and Georgios Paliouras. 2015. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering* 27, 4 (2015), 895–908.
[18] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. 2012. Logic-based event recognition. *The Knowledge Engineering Review* 27, 4 (2012), 469–506.
[19] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*.
[20] Jean Berstel. 2013. *Transductions and context-free languages*. Springer-Verlag.
[21] Henrik Björklund and Thomas Schwentick. 2010. On notions of regularity for data languages. *Theoretical Computer Science* 411, 4 (2010), 702–715. https://doi.org/10.1016/j.tcs.2009.10.009 Fundamentals of Computation Theory.
[22] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-Variable Logic on Data Words. *ACM Trans. Comput. Logic* 12, 4, Article 27 (July 2011), 26 pages. https://doi.org/10.1145/1970398.1970403
[23] Alejandro Buchmann and Boris Koldehofe. 2009. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* (2009).
[24] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. 2003. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.* 14, 6 (2003), 1007–1018.
[25] Benjamin Carle and Paliath Narendran. 2009. On Extended Regular Expressions. In *LATA 2009 (Lecture Notes in Computer Science, Vol. 5457)*. 279–289.
[26] Jan Carlson and Björn Lisper. 2010. A resource-efficient event algebra. *Science of Computer Programming* (2010).
[27] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*.
[28] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2010. A logic-based, reactive calculus of events. *Fundamenta Informaticae* 105, 1-2 (2010), 135–161.
[29] Gianpaolo Cugola and Alessandro Margara. 2009. Raced: an adaptive middleware for complex event detection. In *Middleware*.
[30] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *DEBS*.
[31] Gianpaolo Cugola and Alessandro Margara. 2012. Complex Event Processing with T-REX. *The Journal of Systems and Software* (2012).
[32] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* (2012).
[33] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2005. *A general algebra and implementation for monitoring event streams*. Technical Report. Cornell University.
[34] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards expressive publish/subscribe systems. In *EDBT*.
[35] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. 1989. Making data structures persistent. *Journal of computer and system sciences* 38, 1 (1989), 86–124.
[36] Arnaud Durand and Etienne Grandjean. 2007. First-order queries on structures of bounded degree are computable with constant delay. *ACM Transactions on Computational Logic (TOCL)* 8, 4 (2007), 21–es.
[37] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51. https://doi.org/10.1145/2699442
[38] Antony Galton and Juan Carlos Augusto. 2002. Two approaches to event definition. In *DEXA*.
[39] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352. https://doi.org/10.1007/s00778-019-00557-w
[40] Lukasz Golab and M Tamer Özsu. 2003. Issues in data stream management. *Sigmod Record* (2003).

[41] Alejandro Grez, Cristian Riveros, and Martín Ugarte. 2019. A Formal Framework for Complex Event Processing. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*. 5:1–5:18.
[42] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2020. On the Expressiveness of Languages for Complex Event Recognition. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark (LIPIcs, Vol. 155)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:17. https://doi.org/10.4230/LIPIcs.ICDT.2020.15
[43] Mikell P Groover. 2007. *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall.
[44] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. 2008. On supporting kleene closure over event streams. In *ICDE 2008*. IEEE, 1391–1393.
[45] Yeye He, Siddharth Barman, and Jeffrey F. Naughton. 2014. On Load Shedding in Complex Event Processing. In *ICDT*. 213–224.
[46] Yeye He, Siddharth Barman, Di Wang, and Jeffrey F Naughton. 2011. On the complexity of privacy-preserving complex event processing. In *PODS*. 165–174.
[47] Elena Ikonomovska and Mariano Zelke. 2013. Algorithmic Techniques for Processing Data Streams. *Dagstuhl Follow-Ups* (2013).
[48] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363. https://doi.org/10.1016/0304-3975(94)90242-9
[49] Leonid Libkin. 2013. *Elements of finite model theory*. Springer Science & Business Media.
[50] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*. 889–900.
[51] D Luckham. 1996. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events.
[52] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. 2017. StreamQRE: Modular Specification and Efficient Evaluation of Quantitative Queries over Streaming Data. *SIGPLAN Not.* 52, 6 (June 2017). https://doi.org/10.1145/3140587.3062369
[53] Yuan Mei and Samuel Madden. 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*. ACM, 193–206.
[54] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. 1994. Network intrusion detection. *IEEE network* (1994).
[55] Peter Pietzuch, Brian Shand, and Jean Bacon. 2003. A framework for event composition in distributed systems. In *Middleware*.
[56] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Jousselme. 2019. Composite Event Recognition for Maritime Monitoring. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems* (Darmstadt, Germany) *(DEBS '19)*. ACM, 163–174. https://doi.org/10.1145/3328905.3329762
[57] BS Sahay and Jayanthi Ranjan. 2008. Real time business intelligence in supply chain analytics. *Information Management & Computer Security* (2008).
[58] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS*.
[59] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*. 10–20.
[60] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *ICDT 2013*. 10–20.
[61] Margus Veanes. 2013. Applications of symbolic finite automata. In *CIAA*.
[62] Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. 2007. What is next in event processing?. In *PODS*. 263–272.
[63] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *SIGMOD*.
[64] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*.
[65] Detlef Zimmer and Rainer Unland. 1999. On the semantics of complex events in active database management systems. In *ICDE*.

A PROOF OF LEMMA 7.1

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, $C_1 = C_2$.

Transitivity. 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., $C_i^{<l_j} = \{x \mid x \in C_i \wedge x < l_j\}$. It is clear that $C_1^{<l_1} = C_2^{<l_1}$ and $C_2^{<l_2} = C_3^{<l_2}$, because of (1) and (2), respectively.

Consider first the case where $l_1 < l_2$. This means that (3) $C_1^{<l_1} = C_3^{<l_1}$. 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.

B PROOF OF PROPOSITION 8.12

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 φ_1. The case when $\varphi_1 = \psi$ `IN` A for some ψ 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 ρ and variables A, B we write $\rho^{A \to B}$ to refer to the formula ρ after replacing every occurrence of A by B.

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

- If $\varphi' = \rho_1$ `OP` ρ_2 with `OP` $\in \{;, :, \texttt{OR}\}$, then $\varphi \equiv (\rho_1$ `IN` $A)$ `OP` $(\rho_2$ `IN` $A)$.
- If $\varphi' = \rho$ `OP` with `OP` $\in \{+, \oplus\}$, then $\varphi \equiv (\rho$ `IN` $A)$ `OP`.

- If $\varphi' = \rho$ FILTER $P(\bar{A})$, then
  - if A is not in $\bar{A}$ then $\varphi \equiv (\rho$ IN $A)$ FILTER $P(\bar{A})$, and
  - if A is in $\bar{A}$ then $\varphi \equiv (\rho^{A \to A'}$ IN $A)$ 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 $\varphi = \varphi_1 : \varphi_2$ with φ_1 and φ_2 in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$, there exists a formula ψ in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$ equivalent to φ. By the previous discussion, we can assume that every IN operator in φ is applied at the lowest level. Then the proof follows by doing induction over the structure of φ. The base case is when $\varphi = (R$ IN $A_1 \ldots$ IN $A_j) : (T$ IN $B_1 \ldots$ IN $B_k)$ for some $R, T, A_1, \ldots, A_j, B_1, \ldots, B_k$. Clearly, φ is equal to $\psi =$ STRICT$((R$ IN $A_1 \ldots$ IN $A_j)\,;\,(T$ IN $B_1 \ldots$ IN $B_k))$. For the inductive step, we consider each case separately:

- If $\varphi_1 = \rho_1\,;\,\rho_2$, then $\varphi \equiv \rho_1\,;\,(\rho_2 : \varphi_2)$ and $\rho_2 : \varphi_2$ is smaller than $\varphi_1 : \varphi_2$. By induction hypothesis, $(\rho_2 : \varphi_2)$ has an equivalent formula σ in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$. Thus, $\psi = \rho_1\,;\,\sigma$ is equivalent to φ.
- If $\varphi_1 = \rho$ FILTER $P(\bar{A})$, then $\varphi \equiv (\rho^{\bar{A} \to \bar{A}'} : \varphi_2)$ 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, $(\rho^{\bar{A} \to \bar{A}'} : \varphi_2)$ has an equivalent formula σ in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$. Thus, $\psi = \sigma$ FILTER $P(\bar{A}')$ is equivalent to φ. Note that, since we renamed the variables $\bar{A}$ with $\bar{A}'$ in ρ, 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 $\varphi_1 = \rho_1$ OR ρ_2, then $\varphi \equiv (\rho_1 : \varphi_2)$ OR $(\rho_2 : \varphi_2)$. By induction hypothesis, both $(\rho_1 : \varphi_2)$ and $(\rho_2 : \varphi_2)$ have equivalent formulas σ_1 and σ_2, respectively, in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$. Thus, $\psi = \sigma_1$ OR σ_2 is equivalent to φ.
- If $\varphi_1 = \rho+$, then $\varphi \equiv (\rho : \varphi_2)$ OR $(\rho+\,;\,(\rho : \varphi_2))$. By induction hypothesis, $(\rho : \varphi_2)$ has an equivalent formula σ in CEL$^\star \cup \{$STRICT$\}(\mathcal{P})$. Thus, $\psi = \sigma$ OR $(\rho+\,;\,\sigma)$ is equivalent to φ.

Note that $\varphi_1 = \rho_1$ IN A was not considered, given that IN is included in the base case, by the first part of this proof. Finally, we should also consider when $\varphi_1 = (R$ IN $A_1 \ldots$ IN $A_j)$. For this, we can apply the induction step over φ_2 but these cases are analogous to the previous ones.

The expressive containment is strict: in CEL$^\star \cup \{$STRICT$\}$ the combination 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 CEL$^\star \cup \{:\}$ each : verifies at most one strict concatenation. For instance, formula $\varphi_{\text{STR+}} =$ STRICT$(R+)$ has no equivalent formula in CEL$^\star \cup \{:\}$. We formalize this by defining a property that every formula φ in CEL$^\star \cup \{\,:\,\}$ must satisfy, which basically says that the number of strict-concatenations that φ can verify is bounded by the number of : operations it has.

Given a stream $S = t_0 t_1 \ldots$, an $\mathcal{R}$-tuple g, a valuation μ and a position i with $\min(\sup(\mu)) \leq i < \max(\sup(\mu))$, we define the *insertion* of g at i inside (S, μ) as the pair (S', μ') where:

- S' is result of adding t at position i of S, i.e., $= t_0 t_1 \ldots t_{i-1} g t_i t_{i+1} \ldots$;
- μ' is the result of moving positions of μ accordingly, that is, for every variable A, $\mu(A) = \{j \mid j \in \mu(A) \wedge j < i\} \cup \{j+1 \mid j \in \mu(A) \wedge j \geq i\}$.

A CEL formula φ 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 $\mu \in [\![\varphi]\!](S, p_1, p_2)$ with $|\sup(\mu)| \geq N$, there exists some $i < p_2$ such that $\mu' \in [\![\varphi]\!](S', p_1, p_2 + 1)$, where (S', μ') is the insertion of t at i inside (S, μ). Roughly speaking, if the formula φ 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 the formula φ.

Lemma B.1. *Every formula φ in CEL$^\star \cup \{:\}$ is strict-bounded.*

Now, we use Lemma B.1 to prove that there is no formula in CEL$^\star \cup \{:\}$ equivalent to $\varphi_{\text{STR}+}$. By contradiction, assume there exists such formula, call it ψ. By Lemma B.1, ψ is strict-bounded, so let N be the strict-bounded constant for ψ. Now, consider the stream $S = RR\ldots$ and define μ 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 μ. Because ψ is strict-bounded, there exists some $i < N$ such that $\mu' \in [\![\varphi]\!](S', 0, N+1)$, where (S', μ') is the insertion of t at i inside (S, μ). But then, since we added t in the middle of S and swapped some positions of μ', $\sup(\mu')$ 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 ψ are equivalent.

C PROOF OF PROPOSITION 8.13

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

- If $\varphi' = \rho \texttt{ IN } A$, then $\texttt{STRICT}(\varphi') \equiv \texttt{STRICT}(\rho) \texttt{ IN } A$. By induction hypothesis, there is a formula σ in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$ equivalent to $\texttt{STRICT}(\rho)$. Thus, $\psi' = \sigma \texttt{ IN } A$ is equivalent to $\texttt{STRICT}(\varphi')$.
- If $\varphi' = \rho_1 ; \rho_2$, then $\texttt{STRICT}(\varphi') \equiv \texttt{STRICT}(\rho_1) : \texttt{STRICT}(\rho_2)$. Both $\texttt{STRICT}(\rho_1)$ and $\texttt{STRICT}(\rho_2)$ have equivalent formulas σ_1 and σ_2, respectively, in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$. Thus, $\psi' = \sigma_1 : \sigma_2$ is equivalent to $\texttt{STRICT}(\varphi')$.
- If $\varphi' = \rho \texttt{ FILTER } P$, then $\texttt{STRICT}(\varphi') \equiv \texttt{STRICT}(\rho) \texttt{ FILTER } P$. By induction hypothesis, $\texttt{STRICT}(\rho)$ has an equivalent formula σ in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$. Thus, $\psi' = \sigma \texttt{ FILTER } P$ is equivalent to $\texttt{STRICT}(\varphi')$.
- If $\varphi' = \rho_1 \texttt{ OR } \rho_2$, then $\texttt{STRICT}(\varphi') \equiv \texttt{STRICT}(\rho_1) \texttt{ OR } \texttt{STRICT}(\rho_2)$. Again, both $\texttt{STRICT}(\rho_1)$ and $\texttt{STRICT}(\rho_2)$ have equivalent formulas σ_1 and σ_2, respectively, in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$. Thus, $\psi' = \sigma_1 \texttt{ OR } \sigma_2$ is equivalent to $\texttt{STRICT}(\varphi')$.
- If $\varphi' = \rho+$, then $\texttt{STRICT}(\varphi') \equiv \texttt{STRICT}(\rho)\oplus$. By induction hypothesis, $\texttt{STRICT}(\rho)$ has an equivalent formula σ in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$. Thus, $\psi' = \sigma\oplus$ is equivalent to $\texttt{STRICT}(\varphi')$.

It is left to replace every subformula $\texttt{STRICT}(\varphi')$ of φ with its CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$ equivalent ψ', and the resulting formula will be in CEL$^\star \cup \{:, \oplus\}(\mathcal{P})$ and will be equivalent to φ.

D PROOF OF THEOREM 8.14

Consider $\mathcal{P} = \{P^{\text{SO}}_{=}\}$, where $P_{=}(x, y) := (x.a = y.a)$, and consider the formula:

$$\varphi \;=\; ((A\,;\, E) \texttt{ FILTER } P^{\text{SO}}_{=}(A, E))\oplus$$

in CEL$^\star \cup \{\oplus\}(\mathcal{P})$. We prove that there is no formula ψ in CEL$^\star \cup \{\texttt{STRICT}\}(\mathcal{P})$ equivalent to φ. The strategy of our proof will be to define an ad-hoc "pumping lemma" for CEL$^\star \cup \{\texttt{STRICT}\}$ formulas, to then show that φ does not satisfy such a lemma.

Consider a stream S, a valuation μ, 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 μ 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, μ) k times as a tuple (S', μ'), where S' and μ' 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., $S' = S_1 \cdot S_2 \cdot S_2 \cdot \ldots \cdot S_2 \cdot S_3$.
- μ' is defined as $\mu_1 \cup \mu_2^1 \cup \mu_2^2 \cup \ldots \cup \mu_2^k \cup \mu_3'$, where each μ_2^i is the same valuation μ_2 but with its values moved to fit the i-th occurrence of S_2 in S'. For example, for every variable A, $\mu_2^2(A)$ 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, μ_3' is the same as μ_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 ρ in CEL* is said to be *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 $\mu' \in [\![\rho]\!](S', p_1, p_2')$, where (S', μ') is the results of pumping the fragment $[i, j]$ of (S, μ) 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.

Lemma D.1. *Every formula φ in CEL$^\star \cup \{\texttt{STRICT}\}(\mathcal{P})$ is pumpable.*

Proof. Consider a formula φ in CEL$^\star \cup \{\texttt{STRICT}\}(\mathcal{P})$. We prove the lemma by induction over the structure of φ. First, consider the base case R. By defining $N = 1$ we know that for every stream S there is no valuation $\mu \in [\![\varphi]\!](S)$ with $|\sup(\mu)| > N$, so the lemma holds.

Now, for the inductive step consider first the case $\varphi = \psi_1$ FILTER $P(X_1, \ldots, X_n)$. By induction hypothesis, we know that the lemma holds for ψ_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 $\mu \in [\![\varphi]\!](S, p_1, p_2)$ with $|\sup(\mu)| > N$. By definition $\mu \in [\![\psi_1]\!](S, p_1, p_2)$ and $(S[\mu(X_1)], \ldots, S[\mu(X_n)]) \in P$. By induction hypothesis, ψ_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', μ'), for an arbitrary k. Then, it holds that $\mu' \in [\![\psi_1]\!](S' p_1, p_2')$. 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, $\mu' \in [\![\varphi]\!](S' p_1, p_2')$, thus φ is pumpable.

Consider now the case $\varphi = \psi_1$ OR ψ_2. By induction hypothesis, we know that the property holds for ψ_1 and ψ_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 $\mu \in [\![\varphi]\!](S, p_1, p_2)$ with $|\sup(\mu)| > N$. By definition or OR, either $\mu \in [\![\psi_1]\!](S, p_1, p_2)$ or $\mu \in [\![\psi_2]\!](S, p_1, p_2)$, so w.l.o.g. consider the former case. By induction hypothesis, ψ_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', μ') satisfies $\mu' \in [\![\psi_1]\!](S', p_1, p_2')$. This means that $\mu' \in [\![\varphi]\!](S', p_1, p_2')$, therefore, φ is pumpable.

Now, consider the case $\varphi = \psi_1 \,;\, \psi_2$. By induction hypothesis, we know that the property holds for ψ_1 and ψ_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 $\mu \in [\![\varphi]\!](S, p_1, p_2)$ with $|\sup(\mu)| > N$. This means that there exist $p' \in \mathbb{N}$ and valuations μ_1, μ_2 such that $\mu = \mu_1 \cup \mu_2$, $\mu_1 \in [\![\psi_1]\!](S, p_1, p')$ and $\mu_2 \in [\![\psi_2]\!](S, p'+1, p_2)$. 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, ψ_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', μ_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 μ_2' is the same as μ_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, φ is pumpable.

Consider then the case $\varphi = \psi_1+$. By induction hypothesis, we know that the lemma holds for ψ_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', μ') be the result of pumping the fragment $[i, j]$ of (S, μ) 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 μ' is the same as μ 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 $S_1 \cdot P_0 \cdot S_2 \cdot P_1 \cdot S_2 \cdot \ldots \cdot S_2 \cdot P_k \cdot S_3$. Similarly, μ' is defined as $\mu_1 \cup \mu_2^1 \cup \mu_2^2 \cup \ldots \cup \mu_2^k \cup \mu_3'$, where $\mu_1(A) = \mu_3(A) = \emptyset$ for any variable A, and each μ_2^i is the same valuation μ 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 $\mu_2' = \mu_2^2 \cup \ldots \cup \mu_2^k$ (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 $\mu_2^1 \in [\![\varphi]\!](S', p_1, r)$, because it represents the same valuation as the original one μ. Then, because $\mu' = \mu_2^1 \cup \mu_2'$, 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 $\varphi = \texttt{STRICT}(\psi_1)$. By induction hypothesis, we know that the lemma holds for ψ_1, thus let N_1 be its corresponding constant. Let the constant N for φ 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', μ') satisfies $\mu' \in [\![\psi_1]\!](S, p_1, p_2')$. Notice that $\sup(\mu)$ induces a contiguous interval because of the definition of STRICT, therefore $\sup(\mu')$ also induces a contiguous interval, thus $\mu' \in [\![\varphi]\!](S, p_1, p_2')$.

Finally, consider the case $\varphi = \psi_1\ \texttt{IN}\ A$. By induction hypothesis, we know that the lemma holds for ψ_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, by definition there exists η such that $\eta \in [\![\psi_1]\!](S, p_1, p_2)$ and $\mu = \eta[A \to \sup(\eta)]$. By induction hypothesis there exist positions $i, j \in \sup(\eta)$ such that the fragment $[i, j]$ can be pumped and the result (S', η') satisfies $\eta' \in [\![\psi_1]\!](S, p_1, p_2')$. Note that the results (S', μ') and (S', η') of pumping $[i, j]$ in (S, μ) and (S, η), respectively, are the same, with the only difference that μ' satisfies $\mu'(A) = \sup(\eta')$. Then it follows that $\mu' \in [\![\varphi]\!](S, p_1, p_2')$. □

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

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

Where the first and second lines are the type and a attribute of each event, respectively, and consider the valuation μ 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 μ'. We will analyse the following possible cases: $\text{type}(t_1) = \text{type}(t_2)$; $\text{type}(t_1) = A$ and $\text{type}(t_2) = E$; $\text{type}(t_1) = E$ and $\text{type}(t_2) = A$. In the first case the resulting μ' is a valuation with two consecutive tuples of the same type, which contradicts the original formula φ. In the second case $\sup(\mu_2)$ is not a contiguous interval so the valuation μ' 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 φ. 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 ψ cannot exist.

E PROOF OF PROPOSITION 8.15

Consider a formula φ in CEL $\cup\{:, \oplus\}(\mathcal{U})$. We first prove that there is a ψ in CEL $\cup\{:, \texttt{STRICT}\}(\mathcal{U})$ which is equivalent to φ, and then the proof follows directly from Proposition 8.12.

Consider any formula φ' in CEL $\cup\{:, \texttt{STRICT}\}(\mathcal{U})$. We prove by induction that there exists a formula ψ' in CEL $\cup\{:, \texttt{STRICT}\}(\mathcal{U})$ 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 σ FILTER $P(A)$ with the following procedure:

(1) Push down variable A as shown in the proof of Proposition 8.12, and
(2) Remove the filter and replace each subformula of σ with the form R IN A by R FILTER $P(R)$.

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

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

- For the base case, if $\varphi' = R$, then $\psi' = \texttt{STRICT}(R+)$ is equivalent to $\varphi'\oplus$. Similarly for the case $\varphi' = R$ FILTER $P(R)$.
- If $\varphi' = \texttt{STRICT}(\varphi_1)$, then $\psi' = \texttt{STRICT}(\varphi_1+)$ is equivalent to $\varphi'\oplus$.
- If $\varphi' = \varphi_1 ; \varphi_2$, then $\psi' = \varphi_1 ; (\varphi_2$ OR $((\varphi_2 : \varphi_1)+ ; \varphi_2))$ is equivalent to $\varphi'\oplus$.
- If $\varphi' = \varphi_1+$, then $\psi' = \varphi_1+$ is equivalent to $\varphi'\oplus$.

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

The last and more complex operator is the OR, for which we have to consider $\varphi' = \varphi_1$ OR φ_2, with all possible cases for φ_1 and φ_2. The simplest scenario is where both φ_1 and φ_2 have either the form R, R FILTER $P(R)$ or $\texttt{STRICT}(\psi)$ for some ψ, at which case we can simply write $\varphi'\oplus$ as $\texttt{STRICT}(\varphi'+)$.

Now we consider the cases where some of them does not have this form (w.l.o.g. assume is φ_2). Consider first the case $\varphi_2 = \rho_1 ; \rho_2$. Here we use the following equivalence:

$$
\begin{aligned}
(\varphi_1 \text{ OR } (\rho_1 ; \rho_2))\oplus \quad \equiv \quad & \varphi_1 \oplus \text{ OR } (\rho_1 ; \rho_2) \oplus \text{ OR} && (1)\\
& (\varphi_1 \oplus : (\rho_1 ; \rho_2)\oplus) \oplus \text{ OR} && (2)\\
& \varphi_1 \oplus : ((\rho_1 ; \rho_2) \oplus : \varphi_1\oplus) \oplus \text{ OR} && (3)\\
& ((\rho_1 ; \rho_2) \oplus : \varphi_1\oplus) \oplus \text{ OR} && (4)\\
& (\rho_1 ; \rho_2) \oplus : (\varphi_1 \oplus : (\rho_1 ; \rho_2))\oplus && (5)
\end{aligned}
$$

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) (replacing $(\rho_1 ; \rho_2)\oplus$ by $\rho_1 ; (\rho_2$ 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 $\sigma_2 = \rho_2$ OR $((\rho_2 : \rho_1)+ ; \rho_2)$.

Then, we can replace $(\sigma_1 ; \sigma_2)$ with $\sigma_1 ; (\sigma_2 \texttt{ OR } ((\sigma_2 : \sigma_1)+ ; \sigma_2))$, 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: $\rho+ \equiv \rho \texttt{ OR } \rho ; \rho+$. Thus, φ' can then be written as $(\varphi_1 \texttt{ OR } \rho) \texttt{ OR } (\rho ; \rho+)$. Then, if we redefine $\varphi_1 := (\varphi_1 \texttt{ OR } \rho)$ and $\varphi_2 = (\rho ; \rho+)$, clearly φ' would have the form $\varphi_1 \texttt{ OR } (\rho_1 ; \rho_2)$. 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 φ with its equivalent formula ψ' in a bottom-up fashion to ensure that each φ' is in $\text{CEL} \cup \{ : , \texttt{STRICT}\}(\mathcal{U})$. Finally, the remaining formula ψ does not contain $\oplus$ and thus is in $\text{CEL} \cup \{ : , \texttt{STRICT}\}(\mathcal{U})$. The converse case follows directly from Proposition 8.13.