A formal framework for Complex Event Processing

Alejandro Grez[1], Cristian Riveros[1], and Martín Ugarte[2]

1 **Pontificia Universidad Católica de Chile**
{ajgrez, cristian.riveros}@uc.cl
2 **Université Libre de Bruxelles**
martin.ugarte@ulb.ac.be

Abstract

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

In this paper we embark on the task of giving a rigorous and efficient framework to CEP. 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. 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 studying the syntactical properties of CEL and propose rewriting optimization techniques for simplifying the evaluation of formulas. Then, we introduce a formal computational model for CEP, 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 constant-delay enumeration of the results. Finally, we gather the main results of this work to present an efficient and declarative framework for CEP.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Complex Event Processing (CEP) has emerged as the unifying field of technologies for detecting situations of interest under high-throughput data streams. In scenarios like Network Intrusion Detection [39], Industrial Control Systems [29] or Real-Time Analytics [42], CEP systems aim to efficiently process arriving data, giving timely insights for implementing reactive responses to complex events. Prominent examples of CEP systems from academia and industry include SASE [49], EsperTech [1], Cayuga [26], TESLA/T-Rex [22, 23], among others (see [24] for a survey). The main focus of these systems has been in practical issues like scalability, fault tolerance, and distribution, with the objective of making CEP systems applicable to real-life scenarios. Other design decisions, like query languages, are generally adapted to match computational models that can efficiently process data (see for example [50]). This has produced new data management and optimization techniques, generating promising results in the area [49, 1].

Unfortunately, as has been claimed several times [27, 51, 22, 11] CEP query languages lack a simple and denotational semantics, which makes them difficult to understand, extend or generalize. Their semantics are generally defined either by examples [36, 4, 21], or by intermediate computational models [49, 44, 40]. Although there are frameworks that introduce formal semantics (e.g. [26, 15, 7, 22, 8]), they do not meet the expectations to pave

the foundations of CEP languages. For instance, some of them have unintuitive behavior (e.g. sequencing is non-associative), or are severely restricted (e.g. nesting operators is not supported). One symptom of this problem is that iteration, which is fundamental in CEP, 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 or reuse of variables [49, 26]. As a result of these problems, CEP languages are generally cumbersome.

The lack of simple denotational semantics makes query languages also difficult to evaluate. A common factor in CEP systems is to find sophisticated heuristics [50, 22] that cannot be replicated in other frameworks. Further, optimization techniques are usually proposed at the architecture level [37, 26, 40], which does not allow for a unifying optimization theory. Many CEP frameworks use automata-based models [26, 15, 7] for query evaluation, but these models are usually complicated [40, 44], informally defined [26] or non-standard [22, 5]. In practice this implies that, although finite state automata is a recurring approach in CEP, there is no general evaluation strategy with clear performance guarantees.

Given this scenario, the goal of this paper is to give solid foundations to CEP systems in terms of query language and query evaluation. Towards these goals, we first provide a formal language that allows for expressing the most common features of CEP systems, namely sequencing, filtering, disjunction, and iteration. We introduce complex event logic (CEL), a logic with well-defined compositional and denotational semantics. We also formalize the so-called *selection strategies*, an important notion of CEP that is usually discussed directly [50, 26] or indirectly [15] in the literature but has not been formalized at the language level.

We then focus on the evaluation of CEL. We propose a formal evaluation framework that considers three building blocks: (1) syntactic techniques for rewriting CEL queries, (2) a well-defined intermediate evaluation model, and (3) efficient translation and algorithms to evaluate this model. Regarding the rewriting techniques, we introduce the notions of well-formed and safe formulas in CEL, and show that these restrictions are relevant for query evaluation. Further, we give a general result on rewriting CEL formulas into the so-called LP-normal form, a normal form for dealing with unary filters. For the intermediate evaluation model, we introduce a formal computational model for the regular fragment of CEL, called *complex event automata* (CEA). We show that this model is closed under I/O-determinization and provide translations for CEL formulas with unary filters into CEA. More important, we show an efficient algorithm for evaluating CEA with clear performance guarantees: constant time per tuple followed by constant-delay enumeration of the output. Finally, we bring together our results to present a formal framework for evaluating CEL.

Related work. Active Database Management Systems (ADSMS) and Data Stream Management Systems (DSMS) process data streams, and thus they are usually associated with CEP systems. Both technologies aim to execute relational queries over dynamic data [19, 2, 9]. In contrast, CEP systems see data streams as a sequence of events where the arrival order is the guide for finding patterns inside streams (see [24] for a comparison between ADSMS, DSMS, and CEP). Therefore, DSMS query languages (e.g. CQL [10]) are incomparable with our framework since they do not focus on CEP operators like sequencing and iteration.

Query languages for CEP are usually divided into three approaches [24, 11]: logic-based, tree-based and automata-based models. Logic-based models have their roots in temporal logic or event calculus, and usually have a formal, declarative semantics [8, 12, 20] (see [13] for a survey). However, this approach does not include iteration as an operator or it does not model the output explicitly. Furthermore, their evaluation techniques rely on logic inference mechanisms which are radically different from our approach. Tree-based models [38, 35, 1] have also been used for CEP but their language semantics is usually non-declarative and

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	...

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

their evaluation techniques are based on cost-models, similar to relational database systems.

Automata-based models are close to what we propose in this paper. Previous proposals (e.g. SASE[5], NextCEP[44], DistCED[40]) do not rely in a denotational semantics; their output is defined by intermediate automata models. This implies that either iteration cannot be nested [5] or its semantics is confusing [44]. Other proposals (e.g. CEDR[15], TESLA[22], PBCED[7]) are defined with a formal semantics but they do not include iteration. An exception is Cayuga[25], but its language does not allow reusing variables and sequencing is non-associative, which results in an unintuitive semantics. Our framework is comparable to these systems, but provides a well-defined language that is compositional, allowing arbitrary nesting of operators. Moreover, we present the first evaluation of CEP queries that guarantees constant time per event and constant-delay enumeration of the output.

Finally, there has been some research in theoretical aspects of CEP, e.g. in axiomatization of temporal models [48], privacy [32], and load shedding [31]. This literature does not study the semantics and evaluation of CEP and, therefore, is orthogonal to our work.

Organization. We give an intuitive introduction to CEP and our framework in Section 2. In Section 3 and 4 we formally present our logic and selection strategies. The syntactic structure of the logic is studied in Section 5. The computational model and compilation of formulas are studied in Section 6. In Section 7 we develop efficient evaluation techniques and in Section 8 we present a framework summarizing our results. Future work is discussed in Section 9. Due to space limitations all proofs are deferred to the appendix.

2 Events in action

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

In a CEP setting, events arrive in a streaming fashion to a system that must detect certain *patterns* [24]. 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. Figure 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.

The patterns to be detected are generally specified by domain experts. 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:

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

This formula is asking for two events, one of type temperature (T) and one of type humidity (H). The events of type temperature and humidity are given names x and y, respectively, and the two events are filtered to select only those pairs (x, y) representing a high temperature followed by a low humidity measured by sensor 0.

What should the evaluation of φ_1 over the stream in Figure 1 return? A first important remark is that event streams are noisy, and one does not expect the events matching a formula to be *contiguous* in the stream. Then, a CEP engine needs to be able to dismiss irrelevant events. The semantics of the *sequencing* operator (;) will thus allow for arbitrary events to occur in between the events of interest. A second remark is that in CEP the set of events matching a pattern, called a *complex event*, is particularly relevant to the end user. Every time that a formula matches a portion of the stream, the final user should retrieve the events that compose that portion of the stream. This means that the evaluation of a formula over a stream should output a set of *complex events*. In our framework, each complex event will be the set of indexes (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. What we expect for 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 Figure 1, we can see that the pairs satisfying these conditions are $\{1, 2\}$, $\{1, 8\}$, and $\{5, 8\}$.

Formula φ_1 illustrates the two most elemental features of CEP, namely *sequencing* and *filtering* [24, 9, 50, 2, 17]. But although it detects a set of possible fires, it restricts 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 CEP engines [24, 9]. To illustrate, we extend φ_1 by allowing events to appear in arbitrary order.

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

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

The previous formulas show how CEP systems raise alerts when a certain complex event occurs. However, from a wider scope the objective of CEP is to retrieve information of interest from streams. For example, assume that we want to see how does temperature change 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) [24, 9, 30] is introduced in most CEP frameworks for solving this problem. This operator introduces many difficulties in the semantics of CEP languages. For example, since events are not required to occur contiguously, the nesting of + is particularly tricky and most frameworks simply disallow this (see [49, 10, 26]). Coming back to our example, the formula for measuring temperatures whenever an increase of humidity is detected by sensor 1 is:

$$\varphi_3 = [H \texttt{ AS } x\,;\, (T \texttt{ AS } y \texttt{ FILTER } y.id = 1)+\,;\, H \texttt{ AS } z] \texttt{ FILTER } (x.hum < 30 \ \wedge z.hum > 60 \wedge x.id = 1 \wedge z.id = 1)$$

Intuitively, variables x and z witness the increase of humidity from less than 30% to more than 60%, and y captures temperature measures between x and z. Note that the filter for y is included inside the + operator. Some frameworks allow to declare variables inside a + and filter them outside that operator (e.g. [49]). Although it is possible to define the semantics

for that syntax, this form of filtering makes the definition of nesting + difficult. Another semantic subtlety of the + operator is the association of y to an event. Given that we want to match the event $(T \texttt{ AS } y \texttt{ FILTER } y.id = 1)$ an unbounded number of times: how should the events associated to y occur in the complex events generated as output? Associating different events to the same variable during evaluation has proven to make the semantics of CEP languages cumbersome. In Section 3, we introduce a semantics that allows nesting + and associate variables (inside + operators) to different events across repetitions.

Let us now explain the evaluation of φ_3 over S (Figure 1). The only two humidity events satisfying the top-most filter are $S[3]$ and $S[7]$, and the events in between that satisfy the inner filter are $S[4]$ and $S[6]$. As expected, $\{3, 4, 6, 7\}$ is part of the output. However, there are other complex events in the output. Since, as discussed, there might be irrelevant events between relevant ones, the semantics of + must allow for *skipping* arbitrary events. This implies that the complex events $\{3, 6, 7\}$ and $\{3, 4, 7\}$ are also part of the output.

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 ($\{3, 6, 7\}$, $\{3, 4, 7\}$ and $\{3, 4, 6, 7\}$), one can easily argue that the largest complex event is more informative since all events are contained in it. The complex events output by φ_1 deserve a more thorough analysis. In this scenario, the pairs 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 CEP systems that restrict the output by using so-called *selection strategies* (see for example [49, 50, 22]). Selection strategies are a fundamental feature of CEP. 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 [49, 50]) 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 [15]), it is usually underspecified [49, 50] or complicates the semantics of other operators [26]. In Section 4, we formally define a set of selection strategies including *next*.

Before formally presenting our framework, we illustrate one more common feature of CEP, namely *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:

$$\varphi_4 = [H \texttt{ AS } x; (T \texttt{ AS } y \texttt{ FILTER } y.id = x.id)+; H \texttt{ AS } z] \\ \texttt{FILTER } (x.hum < 30 \ \wedge z.hum > 60 \wedge x.id = z.id)$$

Notice that here the filters contain the predicates $x.id = y.id$ and $x.id = z.id$ that force all events to have the same id. Although this might seem simple, the evaluation of formulas that correlate events introduces new challenges. Intuitively, φ_4 is more complex because the id of x must be remembered in order to compare it with future incoming events. This behavior is clearly not "regular" and it will not be captured by a finite state model [33, 43]. In this paper, we study and characterize the regular part of CEP-systems. In sections 6 and 8 we focus on formulas without correlation. As we will see, the formal analysis of this fragment already presents non-trivial challenges, which is why we defer the analysis of formulas like φ_4 for future work. It is important to mention that the semantics of our language (including selection strategies) is general and includes binary filters like correlation.

3 A query language for CEP

Having discussed the common operators and features of CEP, we proceed to formally introduce CEL (Complex Event Logic), our pattern language for capturing complex events.

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 tuple of attributes in $\mathbf{A}$ denoted by $\text{att}(R)$. If R is a relation name, then an R-tuple is a function $t : \text{att}(R) \to \mathbf{D}$. The type of an R-tuple t is R, and denote this by $\text{type}(t) = R$. For any relation name R, $\text{tuples}(R)$ denotes the set of all possible R-tuples, i.e., $\text{tuples}(R) = \{t : \text{att}(R) \to \mathbf{D}\}$. Similarly, for any database schema $\mathcal{R}$, $\text{tuples}(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} \text{tuples}(R)$.

Given a schema $\mathcal{R}$, an *$\mathcal{R}$-stream* S is an infinite sequence $S = t_0 t_1 \dots$ where $t_i \in \text{tuples}(\mathcal{R})$. When $\mathcal{R}$ is clear from the context, we refer to S simply as a stream. Given a stream $S = t_0 t_1 \dots$ and a position $i \in \mathbb{N}$, the i-th element of S is denoted by $S[i] = t_i$, and the sub-stream $t_i t_{i+1} \dots$ 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 model (like [48]) for future work.

Let $\mathbf{X}$ be a set of variables. Given a schema $\mathcal{R}$, a predicate of arity n is an n-ary relation P over $\text{tuples}(\mathcal{R})$, i.e. $P \subseteq \text{tuples}(\mathcal{R})^n$. An atom is an expression $P(\bar{x})$ where P is an n-ary predicate and $\bar{x} \in \mathbf{X}^n$. As usual, we express predicates as formulas over attributes, and use $x.a$ to reffer to the attribute a of the tuple represented by x. For example, $P(x) := x.hum < 30$ is an atom and P is the predicate of all tuples that have a humidity attribute of less than 30. We consider that checking if a tuple t is in a predicate P takes time $\mathcal{O}(|t|)$, and that every atom $P(\bar{x})$ has constant size (and thus the size of a formula is independent of the type of predicates). We assume a fixed set of predicates $\mathbf{P}$ (i.e. defined by the CEP system). Moreover, we assume that $\mathbf{P}$ is closed under intersection, union, and complement, and $\mathbf{P}$ contains the predicate $P_R(x) := \text{type}(x) = R$ for checking if a tuple is an R-tuple for every $R \in \mathcal{R}$.

CEL syntax. Now we proceed to give the syntax of what we call the *core* of CEL (core-CEL for short), a logic inspired by the operations described in the previous section. This language contains the most essential CEP features. The set of formulas in core-CEL, or core formulas for short, is given by the following grammar:

$$\varphi := R \texttt{ AS } x \mid \varphi \texttt{ FILTER } P(\bar{x}) \mid \varphi \texttt{ OR } \varphi \mid \varphi ; \varphi \mid \varphi+$$

where R is a relation name, x is a variable in $\mathbf{X}$ and $P(\bar{x})$ is an atom in $\mathbf{P}$. For example, all formulas in Section 2 are CEL formulas. Throughout the paper we use φ FILTER $(P(\bar{x}) \wedge Q(\bar{y}))$ or φ FILTER $(P(\bar{x}) \vee Q(\bar{y}))$ as syntactic sugar for $(\varphi$ FILTER $P(\bar{x}))$ FILTER $Q(\bar{y})$ or $(\varphi$ FILTER $P(\bar{x}))$ OR $(\varphi$ FILTER $Q(\bar{y}))$, respectively. Unlike existing frameworks, we do not restrict the syntax, allowing for arbitrary nesting (in particular of +).

CEL semantics. We proceed to define the semantics of core formulas, for which we need to introduce some further notation. A *complex event* C is defined as a non-empty and finite set of indices. As mentioned in Section 2, a complex event contains the positions of the events that witness the matching of a formula over a stream, and moreover, they are the final output of evaluating a formula over a stream. We denote by $|C|$ the size of C and by $\min(C)$ and $\max(C)$ the minimum and maximum element of C, respectively. Given two complex events C_1 and C_2, $C_1 \cdot C_2$ denotes the *concatenation* of two complex events, that is, $C_1 \cdot C_2 := C_1 \cup C_2$ whenever $\max(C_1) < \min(C_2)$ and is undefined otherwise.

In core-CEL formulas, variables are only used to filter and select particular events, i.e. they are not retrieved as part of the output. As examples in Section 2 suggest, we are

only concerned with finding the events that compose the complex events, and not which position corresponds to which variable. The reason behind this is that the operator + allows for repetitions, and therefore variables under (possibly nested) + operators would have a special meaning, particularly for filtering. This discussion motivates the following definitions. Given a formula φ we denote by $\text{var}(\varphi)$ the set of all variables mentioned in φ (including filters), and by $\text{vdef}(\varphi)$ all variables defined in φ by a clause of the form R `AS` x. Furthermore, $\text{vdef}_+(\varphi)$ denotes all variables in $\text{vdef}(\varphi)$ that are defined outside the scope of all + operators. For example, for $\varphi = (T \texttt{ AS } x\,;\, (H \texttt{ AS } y)+) \texttt{ FILTER } z.id = 1$ we have that $\text{var}(\varphi) = \{x, y, z\}$, $\text{vdef}(\varphi) = \{x, y\}$, and $\text{vdef}_+(\varphi) = \{x\}$. Finally, a valuation is a function $\nu : \mathbf{X} \to \mathbb{N}$. Given a finite set of variables $U \subseteq \mathbf{X}$ and two valuations ν_1 and ν_2, the valuation $\nu_1[\nu_2/U]$ is defined by $\nu_1[\nu_2/U](x) = \nu_2(x)$ if $x \in U$ and by $\nu_1[\nu_2/U](x) = \nu_1(x)$ otherwise.

We are ready to define the semantics of a core-CEL formula φ. Given a complex event C and a stream S, we say that C is in the evaluation of φ over S under valuation ν ($C \in [\![\varphi]\!](S, \nu)$) if one of the following conditions holds:

- $\varphi = R$ `AS` x, $C = \{\nu(x)\}$, and $\text{type}(S[\nu(x)]) = R$.
- $\varphi = \psi$ `FILTER` $P(x_1, \ldots, x_n)$, $C \in [\![\psi]\!](S, \nu)$ and $(S[\nu(x_1)], \ldots, S[\nu(x_n)]) \in P$.
- $\varphi = \psi_1$ `OR` ψ_2 and $C \in [\![\psi_1]\!](S, \nu)$ or $C \in [\![\psi_2]\!](S, \nu)$.
- $\varphi = \psi_1\,;\, \psi_2$ and there are $C_1 \in [\![\psi_1]\!](S, \nu)$ and $C_2 \in [\![\psi_2]\!](S, \nu)$ such that $C = C_1 \cdot C_2$.
- $\varphi = \psi+$ and there exists ν' such that $C \in [\![\psi]\!](S, \nu[\nu'/U])$ or $C \in [\![\psi\,;\, \psi+]\!](S, \nu[\nu'/U])$, where $U = \text{vdef}_+(\psi)$.

There are a couple of important remarks here. First, the valuation ν can be defined over a superset of the variables mentioned in the formula. This is important for sequencing (;) because we require the complex events from both sides to be produced with the same valuation. Second, when we evaluate a subformula of the form $\psi+$, we *carry* the value of variables defined outside the subformula. For example, the subformula $(T \texttt{ AS } y \texttt{ FILTER } y.id = x.id)+$ of φ_4 does not define the variable x. However, from the definition of the semantics we see that x will be *already assigned* (because R `AS` x occurs outside the subformula). This is precisely where other frameworks fail to formalize iteration, as without this construct it is not easy to correlate the variables inside + with the ones outside, as we illustrate with φ_4.

As previously discussed, in core-CEL variables are just used for comparing attributes with `FILTER`, but are not relevant for the final output. In consequence, we say that C belongs to the evaluation of φ over S (denoted $C \in [\![\varphi]\!](S)$) if there is a valuation ν such that $C \in [\![\varphi]\!](S, \nu)$. As an example, the complex events presented in Section 2 are indeed the outputs of φ_1 to φ_3 over the stream in Figure 1.

4 Selection strategies

Matching complex events is a computationally intensive task. As the examples in Section 2 suggest, the main reason behind this is that the amount of complex events can grow exponentially in the size of the stream, forcing systems to process large numbers of *candidate* outputs. In order to speed up the matching processes, it is common to restrict the set of results [18, 49, 50]. Unfortunately, most proposals in the literature restrict outputs by introducing heuristics to particular computational models without describing how the semantics are affected. For a more general approach, we introduce *selection strategies* (or *selectors*) as unary operators over core-CEL formulas. Formally, we define four selection strategies called strict (`STRICT`), next (`NXT`), last (`LAST`) and max (`MAX`). `STRICT` and `NXT` are motivated by previously introduced operators [49] under the name of *strict-contiguity* and *skip-till-next-*

match, respectively. `LAST` and `MAX` are useful selection strategies from a semantic point of view. We define each selection strategy below, giving the motivation and formal semantics.

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 2 detects complex events composed by a temperature above 40 degrees followed by a humidity of less than 25%. As already argued, in general one could expect other events between x and y. However, it could be the case that this pattern is of interest only if the events occur contiguously in the stream, or perhaps the stream has been preprocessed by other means and irrelevant events have been thrown out already. For this purpose, `STRICT` reduces the set of outputs selecting only strictly consecutive complex events. Formally, for any CEL formula φ we have that $C \in [\![\texttt{STRICT}(\varphi)]\!](S, \nu)$ holds if $C \in [\![\varphi]\!](S, \nu)$ and for every $i, j \in C$, if $i < k < j$ then $k \in C$ (i.e., C 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.

NXT. The second selector, `NXT`, is similar to the previously proposed operator *skip-till-next-match* [49]. The motivation behind this operator comes from a heuristic that consumes a stream skipping those events that cannot participate in the output, but matching patterns in a *greedy* manner that selects only the first event satisfying the next element of the query. In [49] the authors gave the definition of this strategy just as

> "*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. As we will see later, this allows to speed up the evaluation process as much as skip-till-next-match while providing clear and intuitive semantics. Let C_1 and C_2 be complex events. The symmetric difference between C_1 and C_2 ($C_1 \vartriangle C_2$) is the set of all elements either in C_1 or C_2 but not in both. We say that $C_1 \leq_{\text{next}} C_2$ if either $C_1 = C_2$ or $\min(C_1 \vartriangle C_2) \in C_2$. For example, $\{5, 8\} \leq_{\text{next}} \{1, 8\}$ since the minimum element in $\{5, 8\} \vartriangle \{1, 8\} = \{1, 5\}$ is 1, which is in $\{1, 8\}$. Note that this is intuitively similar to skip-till-next-match, as we are selecting the first relevant event. An important property is that the $\leq_{\text{next}}$-relation forms a total order among complex events, implying the existence of a minimum and a maximum over any finite set of complex events.

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

We can define now the semantics of `NXT`: for a CEL formula φ we have that $C \in [\![\texttt{NXT}(\varphi)]\!](S, \nu)$ if $C \in [\![\varphi]\!](S, \nu)$ and for every complex event $C' \in [\![\varphi]\!](S, \nu)$, if $\max(C) = \max(C')$ then $C' \leq_{\text{next}} C$. In other words, C must be the $\leq_{\text{next}}$-maximum match among all matches that end in $\max(C)$. In our running example, we have that $\{1, 8\}$ matches $\texttt{NXT}(\varphi_1)$ but $\{5, 8\}$ does not. Furthermore, $\{3, 4, 6, 7\}$ matches $\texttt{NXT}(\varphi_4)$ while $\{3, 4, 7\}$ and $\{3, 6, 7\}$ do not. Note that we compare outputs that have the same final position. This way, complex events are discarded only when there is a *preferred* complex event triggered by the same last event.

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 2), we know that

every complex event in $\texttt{NXT}(\varphi_1)$ 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, $\texttt{LAST}(\varphi_1)$ 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, we show in Section 7 that LAST enjoys the same good 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 $C \in [\![\texttt{MAX}(\varphi)]\!](S,\nu)$ holds iff $C \in [\![\varphi]\!](S,\nu)$ and for all $C' \in [\![\varphi]\!](S,\nu)$, if $\max(C) = \max(C')$ then $C \subseteq C'$. 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, $\texttt{MAX}(\varphi_3)$ 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 $\texttt{NXT}(\varphi_3)$ and $\texttt{LAST}(\varphi_3)$ 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.

We have formally presented the foundations of a language for recognizing complex events, and how to restrict the outputs of this language in meaningful manners. Next we study practical aspects of the CEL syntax that impact how efficiently can formulas be evaluated.

5 Syntactic analysis of CEL

We now study the syntactic form of CEL formulas. We define *well-formed* and *safe* formulas, which are syntactic restrictions that characterize semantic properties of interest. Then, we define a convenient normal form and show that any formula can be rewritten in this form.

Syntactic restrictions of formulas. Although CEL has well-defined semantics, there are some formulas whose semantics can be unintuitive. Consider for example the formula $\varphi_5 = (H \texttt{ AS } x) \texttt{ FILTER } (y.tmp \leq 30)$. Here, x will be naturally bound to the only element in a complex event, but y will not *add* a new position to the output. By the semantics of CEL, a valuation ν for φ_5 must assign a position for y that satisfies the filter, but such position is not restricted to occur in the complex event. Moreover, y is not necessarily bound to any of the events seen up to the last element, and thus a complex event could depend on future events. For example, if we evaluate φ_5 over our running example S (Figure 1), we have that $\{2\} \in [\![\varphi_5]\!](S)$, but this depends on the event at position 6. This means that to evaluate this formula we potentially need to inspect events that occur after all events composing the output complex event have been seen, an arguably undesired situation.

To avoid this problem, we introduce the notion of *well-formed* formulas. As the previous example illustrates, this requires defining where variables are *bound* by a subformula of the form $R \texttt{ AS } x$. The set of bound variables of a formula φ is denoted by $\text{bound}(\varphi)$ and is recursively defined as follows:

$$\begin{aligned}
\text{bound}(R \texttt{ AS } x) &= \{x\} & \text{bound}(\psi \texttt{ FILTER } P(\bar{x})) &= \text{bound}(\psi)\\
\text{bound}(\psi_1 \texttt{ OR } \psi_2) &= \text{bound}(\psi_1) \cap \text{bound}(\psi_2) & \text{bound}(\psi+) &= \varnothing\\
\text{bound}(\psi_1 \,;\, \psi_2) &= \text{bound}(\psi_1) \cup \text{bound}(\psi_2) & \text{bound}(\texttt{SEL}(\psi)) &= \text{bound}(\psi)
\end{aligned}$$

where SEL is any selection strategy. We say that a CEL formula φ is *well-formed* if for every subformula of the form ψ FILTER $P(\bar{x})$ and every $x \in \bar{x}$, there is another subformula ψ_x such that $x \in \text{bound}(\psi_x)$ and ψ is a subformula of ψ_x. This definition allows for including filters with variables defined in a wider scope. For example, formula φ_4 in Section 2 is well-formed although it has the not-well-formed formula $(T$ AS y FILTER $y.id = x.id)+$ as a subformula.

One can argue that it would be desirable to restrict the users to only write well-formed formulas. Indeed, the well-formed property can be checked efficiently by a syntactic parser and users should understand that all variables in a formula must be correctly defined. Given that well-formed formulas have a well-defined variable structure, in the future we restrict our analysis to well-formed formulas.

Another issue for CEL is that the reuse of variables can easily produce unsatisfiable formulas. For example, the formula $\psi = T$ AS $x\,;\, T$ AS x is not satisfiable (i.e. $[\![\psi]\!](S) = \varnothing$ for every S) because variable x cannot be assigned to two different positions in the stream. However, we do not want to be too conservative and disallow the reuse of variables in the whole formula (otherwise formulas like φ_2 in Section 2 would not be permitted). This motivates the notion of *safe* CEL formulas. We say that a CEL formula is *safe* if for every subformula of the form $\varphi_1\,;\,\varphi_2$ it holds that $\text{vdef}_+(\varphi_1) \cap \text{vdef}_+(\varphi_2) = \varnothing$. For example, all CEL formulas in this paper are safe except for the formula ψ above.

The safe notion is a mild restriction to help evaluating CEL, and can be easily checked during parsing time. However, safe formulas are a subclass of CEL and it could be the case that they do not capture the full language. We show that this is not the case. Formally, we say that two CEL formulas φ and ψ are equivalent, denoted by $\varphi \equiv \psi$, if $[\![\varphi]\!](S) = [\![\psi]\!](S)$ for every stream S.

▸ **Theorem 2.** *Given a core-CEL formula φ, there is a safe formula φ' such that $\varphi \equiv \varphi'$ and $|\varphi'|$ is at most exponential in $|\varphi|$.*

By this result, we can restrict our analysis to safe formulas without loss of generality. Unfortunately, we do not know if the exponential size of φ' is unavoidable. We conjecture that this is the case, but we do not know yet the corresponding lower bound.

LP-normal form. Now we study how to rewrite CEL formulas to simplify the evaluation of unary filters. Intuitively, filter operators in a CEL formula can become difficult to handle for a query engine. To illustrate this, consider again formula φ_1 in Section 2. Syntactically, this formula states "find an event x followed by an event y, and then check that they satisfy the filter conditions". However, we would like an execution engine to only consider those events x with $id = 0$ that represent temperature above 40 degrees. Only afterwards the possible matching events y should be considered. In other words, formula φ_1 can be restated as:

$$\varphi_1' = [(T \text{ AS } x) \text{ FILTER } (x.tmp > 40 \wedge x.id = 0)]; \\ [(H \text{ AS } y) \text{ FILTER } (y.hum <= 25 \ \wedge y.id = 0)]$$

This example motivates defining the *locally parametrized* normal form (LP normal form). Let $\mathbf{U}$ be the set of all predicates $P \in \mathbf{P}$ of arity 1 (i.e. $P \subseteq \text{tuples}(\mathcal{R})$). We say that a formula φ is in LP-normal form if, for every subformula φ' FILTER $P(\bar{x})$ of φ with $P \in \mathbf{U}$, it holds that $\bar{x} = \{x\}$ and $\varphi' = R$ AS x for some R and x. In other words, all filters containing unary predicates are applied directly to the definitions of their variables. For instance, formula φ_1' is in LP-normal form while formulas φ_1 and φ_2 are not. Note that non-unary predicates are not restricted, and they can be used anywhere in the formula.

One can easily see that having formulas in LP-normal form would be an advantage for an evaluation engine, because it can *filter out* some events as soon as they arrive. However,

formulas that are not in LP-normal form can still be very useful for declaring patterns. To illustrate this, consider the formula:

$$\varphi_6 = (T \texttt{ AS } x); ((T \texttt{ AS } y \texttt{ FILTER } x.temp \geq 40) \texttt{ OR } (H \texttt{ AS } y \texttt{ FILTER } x.temp < 40))$$

Here, the FILTER operator works like a conditional statement: if the x-temperature is greater than 40, then the following event should be a temperature, and a humidity event otherwise. This type of conditional statements can be very useful, but also hard to evaluate. Fortunately, the next result shows that one can always rewrite a formula into LP-normal form, incurring in the worst case in an exponential blow-up in the size of the formula.

▸ **Theorem 3.** *Let φ be a CEL formula. Then, there is a CEL formula ψ in LP-normal form such that $\varphi \equiv \psi$, and $|\psi|$ is at most exponential in $|\varphi|$.*

The importance of this result and Theorem 2 will become clear in the next sections, where we show that safe formulas in LP-normal form have good properties for evaluation. Similar to Theorem 2, we do not know if the exponential blow-up is unavoidable and leave this for future work.

6 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). Similar to classical database management systems (DBMS), it is useful to have a formal model that stands between the query language and the evaluation algorithms, in order to simplify the analysis and optimization of the whole evaluation process. There are several examples of DBMS that are based on this approach like regular expressions and finite state automata [33, 6], and SQL and relational algebra [3, 41]. Here, we propose CEA as the intermediate evaluation model for CEL and show later how to compile any (unary) CEL formula into a CEA.

As its name suggests, complex event automata (CEA) are an extension of *Finite State Automata* (FSA). The first difference from FSA comes from handling streams instead of words. A CEA is said to run over a stream of tuples, unlike FSA which run over words of a certain alphabet. The second difference arises directly from the first one by the need of processing tuples, which can have infinitely many different values, in contrast to the finite input alphabet of FSA. To handle this, our model is extended the same way as a Symbolic Finite Automata (SFA) [47]. SFAs are finite state automata in which the alphabet is described implicitly by a boolean algebra over the symbols. This allows automata to work with a possibly infinite alphabet and, at the same time, use finite state memory for processing the input. CEA are extended analogously, which is reflected in transitions labeled by unary predicates over tuples. The last difference addresses the need to generate complex events instead of boolean answers. A well known extension for FSA are *Finite State Transducers* [16], which are capable of producing an output whenever an input element is read. Our computational model follows the same approach: CEA are allowed to generate and output complex events when reading a stream.

Recall from Section 5 that $\mathbf{U}$ is the subset of unary predicates of $\mathbf{P}$. Let $\bullet, \circ$ be two symbols. A *complex event automaton* (CEA) is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of states, $\Delta \subseteq Q \times (\mathbf{U} \times \{\bullet, \circ\}) \times Q$ is the transition relation, and $I, F \subseteq Q$ are the set of initial and final states, respectively. Given a stream $S = t_0 t_1 \ldots$, a run ρ of $\mathcal{A}$ over S is a sequence of transitions: $\rho : q_0 \xrightarrow{P_0/m_0} q_1 \xrightarrow{P_1/m_1} \cdots \xrightarrow{P_n/m_n} q_{n+1}$ such that $q_0 \in I$, $t_i \in P_i$ and $(q_i, P_i, m_i, q_{i+1}) \in \Delta$ for every $i \leq n$. We say that ρ is *accepting* if $q_{n+1} \in F$ and $m_n = \bullet$. We

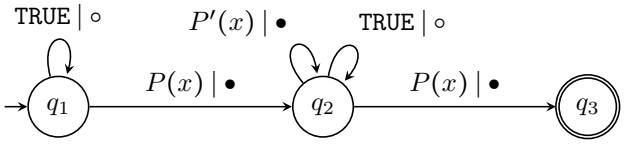


Figure 2 A CEA that can generate an unbounded amount of complex events. Here $P(x) := \text{type}(x) = H$ and $P'(x) := \text{type}(x) = T \wedge x.temp > 40$.

denote by $\text{Run}_n(\mathcal{A}, S)$ the set of accepting runs of $\mathcal{A}$ over S of length n. Further, $\text{events}(\rho)$ is the set of positions where the run *marks* S, namely $\text{events}(\rho) = \{i \in [0, n] \mid m_i = \bullet\}$. Intuitively this means that when a transition is taken, if the transition has the $\bullet$ symbol then the *current* position of the stream is included in the output (similar to the execution of a transducer). Note that we require the last position of an accepting run to be marking, as otherwise an output could depend on *future* events (see the discussion about well-formed formulas in Section 5). Given a stream S and $n \in \mathbb{N}$, we define the set of complex events of $\mathcal{A}$ over S at position n as $[\![\mathcal{A}]\!]_n(S) = \{\text{events}(\rho) \mid \rho \in \text{Run}_n(\mathcal{A}, S)\}$ and the set of all complex events as $[\![\mathcal{A}]\!](S) = \bigcup_n [\![\mathcal{A}]\!]_n(S)$. Note that $[\![\mathcal{A}]\!](S)$ can be infinite, but $[\![\mathcal{A}]\!]_n(S)$ is finite.

Consider as an example the CEA $\mathcal{A}$ depicted in Figure 2. In this CEA, each transition $P(x)|\bullet$ marks one H-tuple and each transition $P'(x)|\bullet$ marks a T-tuple with temperature bigger than 40. Note also that the transitions labeled by $\texttt{TRUE}|\circ$ allow $\mathcal{A}$ to arbitrarily skip tuples of the stream. Then, for every stream S, $[\![\mathcal{A}]\!](S)$ represents the set of all complex events that begin and end with an H-tuple and also contain some of the T-tuples with temperature higher than 40.

It is important to stress that CEA are designed to be an evaluation model for the unary fragment of CEL (a formal definition is presented in the next paragraph). Several computational models have been proposed for complex event processing [26, 40, 49, 44], but most of them are informal and non-standard extensions of finite state automata. In our framework, we want to take a step back compared to previous proposals and define a simple but powerful model that captures the *regular core* of CEL. Intuitively, formulas like φ_1, φ_2 and φ_3 in Section 2 can be evaluated using a bounded amount of memory. In contrast, formula φ_4 needs unbounded memory to store *candidate* events seen in the past, and thus, it calls for a more sophisticated model (e.g. data automata [45]). Of course one would like to have a full-fledged model for CEL, but to this end we must first understand the regular fragment. A computational model for the whole CEP logic is left as future work.

Compiling unary CEL into CEA. We now show how to compile a well-formed and unary CEL formula φ into a CEA $\mathcal{A}_\varphi$. Formally, we say a CEL formula φ is equivalent to a CEA $\mathcal{A}$ if $[\![\varphi]\!](S) = [\![\mathcal{A}]\!](S)$ for every stream S. A CEL formula φ is unary if for every subformula of φ of the form φ' $\texttt{FILTER}$ $P(\bar{x})$, it holds that $P(\bar{x})$ is a unary predicate (i.e. $P(\bar{x}) \in \mathbf{U}$). For example, formulas φ_1, φ_2, and φ_3 in Section 2 are unary, but formula φ_4 is not (the predicate $y.id = x.id$ is binary). As motivated in Section 2 and 5, despite their apparent simplicity unary formulas already present non-trivial computational challenges (see Section 7).

▸ **Theorem 4.** *For every well-formed formula φ in unary core-CEL, there is a CEA $\mathcal{A}_\varphi$ equivalent to φ. Furthermore, $\mathcal{A}_\varphi$ is of size at most linear in $|\varphi|$ if φ is safe and in LP-normal form, and at most double exponential in $|\varphi|$ otherwise.*

The proof of Theorem 4 is closely related with the safeness condition and the LP-normal form presented in Section 5. The construction first converts φ into an equivalent CEL formula φ' in LP-normal form (Theorem 3) and then builds an equivalent CEA from φ'. Unfortunately, there is an exponential blow-up for converting φ into LP-normal form. However, we show

that the output is of linear size if φ' is safe, and of exponential size otherwise, suggesting that restricting the language to safe formulas allows for more efficient evaluation.

We have described the compilation process without considering selection strategies. To include them, we extend our notation and allow selection strategies to be applied over CEA. Given a CEA $\mathcal{A}$, a selection strategy $\texttt{SEL} \in \{\texttt{STRICT}, \texttt{NXT}, \texttt{LAST}, \texttt{MAX}\}$ and stream S, the set of outputs $[\![\texttt{SEL}(\mathcal{A})]\!](S)$ is defined analogously to $[\![\texttt{SEL}(\varphi)]\!](S)$ for a formula φ. Then, we say that a CEA $\mathcal{A}_1$ is equivalent to $\texttt{SEL}(\mathcal{A}_2)$ if $[\![\mathcal{A}_1]\!](S) = [\![\texttt{SEL}(\mathcal{A}_2)]\!](S)$ for every stream S.

▸ **Theorem 5.** *Let* SEL *be a selection strategy. For any CEA $\mathcal{A}$, there is a CEA $\mathcal{A}_{\texttt{SEL}}$ equivalent to* $\texttt{SEL}(\mathcal{A})$*. Furthermore, the size of $\mathcal{A}_{\texttt{SEL}}$ is, with respect to the size of $\mathcal{A}$, at most linear if* SEL = STRICT*, and at most exponential otherwise.*

At first this result might seem unintuitive, specially in the case of NXT, LAST and MAX. It is not immediate (and rather involved) to show that there exists a CEA for these strategies because they need to *track* an unbounded number of complex events using finite memory. Still, this can be done with an exponential blow-up in the number of states.

Theorem 5 concludes our study of the compilation of unary CEL into CEA. We have shown that not only is CEA able to evaluate CEL formulas, but it can also be exploited to evaluate selections strategies. We conclude this section by introducing the notion of I/O-determinism that will be crucial for our evaluation algorithms in the next section.

I/O-deterministic CEA. To evaluate CEA in practice we will focus on the class of the so-called *I/O-deterministic* CEA (for Input/Output deterministic). A CEA $\mathcal{A} = (Q, \Delta, I, F)$ is I/O-deterministic if $|I| = 1$ and for any two transitions (p, P_1, m_1, q_1) and (p, P_2, m_2, q_2), either P_1 and P_2 are mutually exclusive (i.e. $P_1 \cap P_2 = \varnothing$), or $m_1 \neq m_2$. Intuitively, this notion imposes that given a stream S and a complex event C, there is at most one run over S that generates C (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.

I/O-deterministic CEA are important because they allow for a simple and efficient evaluation algorithm (discussed in Section 7). But for this algorithm to be useful, we need to make sure that every CEA can be I/O determinized. Formally, we say that two CEA $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent (denoted $\mathcal{A}_1 \equiv \mathcal{A}_2$) if for every stream S we have $[\![\mathcal{A}_1]\!](S) = [\![\mathcal{A}_2]\!](S)$.

▸ Proposition 1. For every CEA $\mathcal{A}$ there is an I/O-deterministic CEA $\mathcal{A}'$ such that $\mathcal{A} \equiv \mathcal{A}'$, and $\mathcal{A}'$ is of size at most exponential over $|\mathcal{A}|$. That is, CEA are closed under I/O-determinization.

This result and the compilation process allow us to evaluate any CEL formula by means of I/O-deterministic CEA without loss of generality.

7 Algorithms for evaluating CEA

In this section we show how to efficiently evaluate CEA. We start by formalizing the notion of *efficient evaluation* in CEP, which has not been formalized before in the CEP literature.

Efficiency in CEP. Defining a notion of efficiency for CEP is challenging since we would like to compute complex events in one pass and using a restricted amount of resources. Streaming algorithms [34, 28] are a natural starting point as they 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 is that in CEP 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 parts: (1) consuming

new events and updating the internal memory of the system and (2) generating complex events from the internal memory of the system. We require both parts to be as efficient as possible. First, (1) should process each event in a time that does not depend on the number of events seen in the past. Second, (2) should not spend any time *processing* and instead it should be completely devoted to generating the output. To formalize this notion, we assume that there is a special instruction $\texttt{yield}_S$ that returns the next element of a stream S. Then, given a function $f : \mathbb{N} \to \mathbb{N}$, a *CEP evaluation algorithm* with f-update time is an algorithm that evaluates a CEA $\mathcal{A}$ over a stream S such that:

1. between any two calls to $\texttt{yield}_S$, the time spent is bounded by $\mathcal{O}(f(|\mathcal{A}|) \cdot |t|)$, where t is the tuple returned by the first of such calls, and
2. maintains a data structure D in memory, such that after calling $\texttt{yield}_S$ n times, the set $[\![\mathcal{A}]\!]_n(S)$ can be enumerated from D with constant delay.

The notion of constant-delay enumeration was defined in the database community [46, 14] precisely for defining efficiency whenever generating the output might use considerable time. Formally, it requires the existence of a routine ENUMERATE that receives D as input and outputs all complex events in $[\![\mathcal{A}]\!]_n(S)$ without repetitions, while spending a constant amount of time before and after each output. Naturally, the time to generate a complex event C must be linear in $|C|$. We remark that **1.** is a natural restriction imposed in the streaming literature [34], while **2.** is the minimum requirement if an arbitrarily large set of arbitrarily large outputs must be produced [46].

Note that the update time $\mathcal{O}(f(|\mathcal{A}|) \cdot |t|)$ is linear in $|t|$ if we consider that $\mathcal{A}$ is fixed. Since this is the case in practice (i.e. the automaton is generally small with respect to the stream, and does not change during evaluation), this amounts to constant update time when measured under data complexity (tuples can also be considered of constant size).

Efficient evaluation of CEA. Having a good notion of efficiency, we proceed to show how to evaluate CEA efficiently. As it was previously discussed in Section 6, I/O deterministic CEA are specially designed for having CEP evaluation algorithms with linear update time. Furthermore, given that any CEA can be I/O-determinized (Proposition 1), this implies a CEP evaluation algorithm to evaluate any CEA. Unfortunately, the determinization procedure has an exponential blow-up in the size of the automaton, increasing the update time when the automaton is not I/O deterministic.

▸ **Theorem 6.** *For every I/O-deterministic CEA $\mathcal{A}$, there is a CEP evaluation algorithm with $|\mathcal{A}|$-update time. Furthermore, if $\mathcal{A}$ is any CEA, there is a CEP evaluation algorithm with $2^{|\mathcal{A}|}$-update time.*

We can further extend the CEP evaluation algorithm for I/O-deterministic CEA to any selection strategies by using the results of Theorem 5. However, by naively applying Theorem 5 and then I/O-determinizing the resulting automaton, we will have a double exponential blow-up. By doing the compilation of the selection strategies and the I/O-determinization together, we can lower the update time. Moreover, and rather surprisingly, we can evaluate `NXT` and `LAST` without determinizing the automaton, and therefore with linear update time.

▸ **Theorem 7.** *Let* `SEL` *be a selection strategy. For any CEA $\mathcal{A}$, there is a CEP evaluation algorithm for* $\texttt{SEL}(\mathcal{A})$*. Furthermore, the update time is $|\mathcal{A}|$ if* $\texttt{SEL} \in \{\texttt{NXT}, \texttt{LAST}\}$*, $2^{|\mathcal{A}|}$ if* $\texttt{SEL} = \texttt{STRICT}$ *and $4^{|\mathcal{A}|}$ if* $\texttt{SEL} = \texttt{MAX}$*.*

The algorithms of Theorem 6 and 7 are probably the most interesting technical results of the paper and, unfortunately, given space restrictions they are deferred to the appendix.

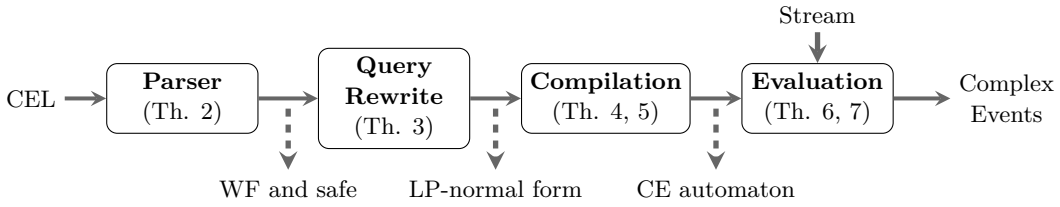


Figure 3 Evaluation framework for CEL.

8 An evaluation framework for CEL

Having all the building blocks, we put all the results in perspectives and show how to evaluate unary CEL formulas. In Figure 3, we show the evaluation cycle of a CEL formula in our framework and how all the results and theorems fit together. To explain this framework, consider a unary CEL formula φ (possibly with selection strategies). The process starts in the parser module, where we check if φ is well-formed and safe. These conditions are important to ensure that φ is satisfiable and make a correct use of variables. Note that a CEP system could translate unsafe formulas (Theorem 2), incurring however in an exponential blow-up.

The next module rewrites a well-formed and safe formula φ into LP-normal form by using the rewriting process of Theorem 3. In the worst case this produces an exponentially larger formula. To avoid this, in many cases one can apply *local rewriting rules* [3, 41]. For example, in Section 2 we converted φ_1 into φ_1' by applying a *filter push*, avoiding the exponential blow-up of Theorem 3. Unfortunately, we cannot apply this over formulas like φ_6 in Section 5. Nevertheless, formulas like φ_6 are rather uncommon in practice and local rewriting rules will usually produce LP-formulas of polynomial size.

The third module receives a formula in LP-normal form and builds a CEA $\mathcal{A}_\varphi$ of polynomial size (Theorem 4 and 5). Then, the last module runs $\mathcal{A}_\varphi$ over the stream by using our CEP evaluation procedure for I/O deterministic CEA (Theorem 6). If there is no selection strategy, $\mathcal{A}_\varphi$ must be determinized before running the CEP evaluation algorithm. In the worst case, this determinization is exponential in $\mathcal{A}_\varphi$, nevertheless, in practice the size of $\mathcal{A}_\varphi$ is rather small. If a selection strategy `SEL` is used, we can use the algorithms of Theorem 7 for evaluating $\texttt{SEL}(\mathcal{A}_\varphi)$, having a similar update time than evaluating $\mathcal{A}_\varphi$ alone. It is worth mentioning that evaluating $\texttt{NXT}(\mathcal{A}_\varphi)$ or $\texttt{LAST}(\mathcal{A}_\varphi)$ has even better performance than evaluating $\mathcal{A}_\varphi$ directly, given that the update time is linear in the size of $\mathcal{A}_\varphi$.

9 Future work

This paper settles new foundations for CEP systems, stimulating new research directions. In particular, a natural next step is to study the evaluation of non-unary CEL formulas. This requires new insight in rewriting formulas and a more powerful computational model with CEP evaluation algorithms. Another relevant problem is to understand the expressive power of different fragments of CEL and the relationship between the different operators. In this same direction, we envision as future work a generalization of the concept behind selection strategies, together with a thorough study of their expressive power.

Finally, we have focused on the fundamental features of CEP languages, leaving other features outside to keep the language and analysis simple. These features include correlation, time windows, aggregation, consumption policies, among others. We plan to extend CEL gradually with these features to establish a more complete and formal framework for CEP.

References

1 Esper enterprise edition website. http://www.espertech.com/. Accessed on 2018-01-05.
2 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. Aurora: A data stream management system. In *SIGMOD*, 2003.
3 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley, 1995.
4 A. Adi and O. Etzion. Amit-the situation manager. *VLDB Journal*, 2004.
5 J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.
6 A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*. 1990.
7 M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *VLDB*, 2008.
8 D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. A rule-based language for complex event processing and reasoning. In *RR*, 2010.
9 A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager (demonstration description). In *SIGMOD*, 2003.
10 A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 2006.
11 A. Artikis, A. Margara, M. Ugarte, S. Vansummeren, and M. Weidlich. Complex event recognition languages: Tutorial. In *DEBS*, pages 7–10. ACM, 2017.
12 A. Artikis, M. Sergot, and G. Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):895–908, 2015.
13 A. Artikis, A. Skarlatidis, F. Portet, and G. Paliouras. Logic-based event recognition. *The Knowledge Engineering Review*, 27(4):469–506, 2012.
14 G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007.
15 R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
16 J. Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.
17 A. Buchmann and B. Koldehofe. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 2009.
18 J. Carlson and B. Lisper. A resource-efficient event algebra. *Science of Computer Programming*, 2010.
19 J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.
20 F. Chesani, P. Mello, M. Montali, and P. Torroni. A logic-based, reactive calculus of events. *Fundamenta Informaticae*, 105(1-2):135–161, 2010.
21 G. Cugola and A. Margara. Raced: an adaptive middleware for complex event detection. In *Middleware*, 2009.
22 G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS*, 2010.
23 G. Cugola and A. Margara. Complex event processing with t-rex. *The Journal of Systems and Software*, 2012.
24 G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 2012.
25 A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. A general algebra and implementation for monitoring event streams. Technical report, Cornell University, 2005.

26 A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

27 A. Galton and J. C. Augusto. Two approaches to event definition. In *DEXA*, 2002.

28 L. Golab and M. T. Özsu. Issues in data stream management. *Sigmod Record*, 2003.

29 M. P. Groover. *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall, 2007.

30 D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE 2008*, pages 1391–1393. IEEE, 2008.

31 Y. He, S. Barman, and J. F. Naughton. On load shedding in complex event processing. In *ICDT*, pages 213–224, 2014.

32 Y. He, S. Barman, D. Wang, and J. F. Naughton. On the complexity of privacy-preserving complex event processing. In *PODS*, pages 165–174, 2011.

33 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 1979.

34 E. Ikonomovska and M. Zelke. Algorithmic techniques for processing data streams. *Dagstuhl Follow-Ups*, 2013.

35 M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.

36 D. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events, 1996.

37 M. Mansouri-Samani and M. Sloman. Gem: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 1997.

38 Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206. ACM, 2009.

39 B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE network*, 1994.

40 P. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In *Middleware*, 2003.

41 R. Ramakrishnan and J. Gehrke. *Database management systems (3 ed.)*. McGraw-Hill, 2003.

42 B. Sahay and J. Ranjan. Real time business intelligence in supply chain analytics. *Information Management & Computer Security*, 2008.

43 J. Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009.

44 N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

45 L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, 2006.

46 L. Segoufin. Enumerating with constant delay the answers to a query. In *ICDT 2013*, pages 10–20, 2013.

47 M. Veanes. Applications of symbolic finite automata. In *CIAA*, 2013.

48 W. White, M. Riedewald, J. Gehrke, and A. Demers. What is next in event processing? In *PODS*, pages 263–272, 2007.

49 E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

50 H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, 2014.

51 D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE*, 1999.

A Proofs of Section 4

A.1 Proof of Lemma 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 $C_2 \leq_{\text{next}} C_1$. $C_2 \leq_{\text{next}} C_1$ means that either $C_1 = C_2$ or (1) $\min(C_1 \triangle C_2) \in C_1$, and $C_1 \leq_{\text{next}} C_2$ that either $C_2 = C_1$ or (2) $\min(C_1 \triangle C_2) \in C_2$. By contradiction, consider that $C_1 \neq C_2$. Then, both (1) and (2) must be true, but because of the 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 $C_2 \leq_{\text{next}} C_3$. Because $C_1 \leq_{\text{next}} C_2$ holds, then either $C_1 = C_2$ or (1) $\min(C_1 \triangle C_2) \in C_2$. If $C_1 = C_2$, then $C_1 \leq_{\text{next}} C_3$ because $C_2 \leq_{\text{next}} C_3$. Now, if $C_1 \neq C_2$, then (1) must hold, which implies that the lowest element that is either in C_1 or C_2, but not in both, has to be in C_2. Let's call this element l_1. Because $C_2 \leq_{\text{next}} C_3$, then either $C_2 = C_3$ or (2) $\min(C_2 \triangle C_3) \in C_3$. Again, if $C_2 = C_3$, then $C_1 \leq_{\text{next}} C_3$ because $C_1 \leq_{\text{next}} C_2$. Now, if $C_2 \neq C_3$, then (2) must hold, so the lowest element that is either in C_2 or C_3, but not in both, has to be in C_3. Let's call this element l_2.

Given that $C_1 \neq C_2$ and $C_2 \neq C_3$, 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. Also, because of (2) it holds that $l_2 \notin C_2$, so $l_1 \neq l_2$.

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 Proofs of Section 5

B.1 Proof of Theorem 2

To prove this theorem, we first show that one can push disjunction (by means of OR) to the top-most level of every core-CEL formula. Formally, we say that a CEL formula φ is in disjunctive-normal form if $\varphi = (\varphi_1 \texttt{ OR } \cdots \texttt{ OR } \varphi_n)$, where for each $i \in \{1, \ldots, n\}$, it is the case that:

- Every OR operator in φ_i occurs in the scope of a + operator.

- For every subformula of φ_i of the form $(\varphi'_i)+$, it is the case that φ'_i is in disjunctive normal form.

Now we show that every formula can be translated into disjunctive normal form.

▸ **Lemma 8.** *Every formula φ in core-CEL can be translated into disjunctive-normal form in time at most exponential $|\varphi|$.*

Proof. We proceed by induction over the structure of φ.

- If $\varphi = R$ AS x, then φ is already free of OR.
- If $\varphi = \varphi_1$ OR φ_2, the result readily follows from the induction hypothesis.
- If $\varphi = (\varphi')+$, by induction hypothesis φ can be translated into disjunctive normal form.
- If $\varphi = \varphi'$ FILTER $P(\bar{x})$ with $\bar{x} = (x_1, \ldots, x_k)$, we know by induction hypothesis that φ' is equivalent to a formula $(\varphi_1$ OR $\cdots$ OR $\varphi_n)$. Therefore, φ is equivalent to the formula $(\varphi_1$ OR $\cdots$ OR $\varphi_n)$ FILTER $P(\bar{x})$. We show that this latter formula is equivalent to $(\varphi_1$ FILTER $P(\bar{x}))$ OR $\cdots$ OR $(\varphi_n$ FILTER $P(\bar{x}))$. Let S be a stream and assume $C \in [\![(\varphi_1$ OR $\cdots$ OR $\varphi_n)$ FILTER $P(\bar{x})]\!](S)$. Then, there is some ν such that $C \in [\![(\varphi_1$ OR $\cdots$ OR $\varphi_n)]\!](S, \nu)$ and $(S[\nu(x_1)], \ldots, S[\nu(x_k)]) \in P(\bar{x})$. By definition of OR, this implies that there is an $i \in \{1, \ldots, n\}$ such that $C \in [\![(\varphi_i)]\!](S, \nu)$. Now, because $(S[\nu(x_1)], \ldots, S[\nu(x_k)]) \in P$, we have $C \in [\![(\varphi_i)$ FILTER $P(\bar{x})]\!](S, \nu)$. We can then immediately conclude that C is in $[\![(\varphi_1$ FILTER $P(\bar{x}))$ OR $\cdots$ OR $(\varphi_n$ FILTER $P(\bar{x}))]\!](S, \nu)$, and therefore that C is in $[\![(\varphi_1$ FILTER $P(\bar{x}))$ OR $\cdots$ OR $(\varphi_n$ FILTER $P(\bar{x}))]\!](S)$. The converse follows from an analogous argument.
- If $\varphi = (\varphi_1 \,;\, \varphi_2)$, by induction hypothesis φ_1 is equivalent to a formula $(\varphi_1^1$ OR $\cdots$ OR $\varphi_n^1)$ and φ_2 is equivalent to a formula $(\varphi_1^2$ OR $\cdots$ OR $\varphi_m^2)$. Let φ' be defined by

$$\varphi' = (\varphi_1^1 \,;\, \varphi_1^2) \text{ OR } (\varphi_1^1 \,;\, \varphi_2^2) \text{ OR } \cdots \text{ OR } (\varphi_1^1 \,;\, \varphi_m^2) \text{ OR } (\varphi_2^1 \,;\, \varphi_1^2) \text{ OR } \cdots \text{ OR } (\varphi_2^1 \,;\, \varphi_m^2) \text{ OR } \cdots \text{ OR } (\varphi_n^1 \,;\, \varphi_1^2) \text{ OR } \cdots \text{ OR } (\varphi_n^1 \,;\, \varphi_m^2).$$

We show that $\varphi \equiv \varphi'$. Let S be a stream and let C be a complex event. If $C \in [\![\varphi]\!](S)$, then there is a valuation ν and two complex events C_1 and C_2 such that $C = C_1 \cdot C_2$, $C_1 \in [\![\varphi_1]\!](S, \nu)$ and $C_2 \in [\![\varphi_2]\!](S, \nu)$. Then, there are two numbers i and j such that $C_1 \in [\![\varphi_i^1]\!](S, \nu)$ and $C_2 \in [\![\varphi_j^2]\!](S, \nu)$. As $C = C_1 \cdot C_2$, it immediately follows that $C \in [\![\varphi_i^1 \,;\, \varphi_j^2]\!](S)$, and thus $C \in [\![\varphi']\!](S)$.
For the converse assume $C \in [\![\varphi']\!](S)$. Then, there is a valuation ν, a complex event C and two numbers i and j such that $C \in [\![\varphi_i^1 \,;\, \varphi_j^2]\!](S, \nu)$. Therefore there are two complex events C_1 and C_2 such that $C = C_1 \cdot C_2$, $C_1 \in [\![\varphi_i^1]\!](S, \nu)$ and $C_2 \in [\![\varphi_j^2]\!](S, \nu)$. By semantics of OR, we have $C_1 \in [\![\varphi_1]\!](S, \nu)$ and $C_2 \in [\![\varphi_2]\!](S, \nu)$. As $C = C_1 \cdot C_2$, it readily follows that $C \in [\![\varphi_1 \,;\, \varphi_2]\!](S) = [\![\varphi]\!](S)$.

◂

Having this result, we proceed to show that a core-CEL formula in disjunctive normal form can be translated into a safe formula. To this end, we need to show the following two lemmas.

▸ **Lemma 9.** *Let φ be a core-CEL formula in which every* OR *occurs inside the scope of a $+$ operator, and let $x \in \text{vdef}_+(\varphi)$. Then, for every complex event C, valuation ν and stream S such that $C \in [\![\varphi]\!](S, \nu)$, it is the case that $x \in \text{dom}(\nu)$ and $\nu(x) \in C$.*

Proof. We proceed by induction on the structure of φ. Let ν be a valuation, S a stream and C a complex event.

- Assume $\varphi = R$ AS x and that $C \in [\![\varphi]\!](S, \nu)$. By definition, we have $C = \{\nu(x)\}$.
- Assume $\varphi = \varphi'$ FILTER $P(\bar{x})$ and that $C \in [\![\varphi]\!](S, \nu)$. Let $x \in \text{vdef}_+(\varphi)$. By definition, we have that $C \in [\![\varphi']\!](S, \nu)$. Since $x \in \text{vdef}_+(\varphi')$, by induction hypothesis we have $x \in \text{dom}(\nu)$ and $\nu(x) \in C$.
- If $\varphi = (\varphi')+$ the condition trivially holds as $\text{vdef}_+(\varphi) = \varnothing$.
- If $\varphi = \varphi_1 ; \varphi_2$, then $x \in \text{vdef}_+(\varphi_1)$ or $x \in \text{vdef}_+(\varphi_2)$. Assume w.l.o.g. that $x \in \text{vdef}_+(\varphi_1)$. If $C \in [\![\varphi]\!](S, \nu)$, then $C = C_1 \cdot C_2$, where $C_1 \in [\![\varphi_1]\!](S, \nu)$. As $x \in \text{vdef}_+(\varphi_1)$, by induction hypothesis we have that $x \in \text{dom}(\nu)$ and $\nu(x) \in C_1 \subseteq C$, concluding the proof.

◂

▸ **Lemma 10.** *Let φ be a core-CEL formula in which every* OR *occurs inside the scope of a + operator, and let S be a stream. If φ has a subformula φ' that is not under the scope of a + operator such that $[\![\varphi']\!](S) = \varnothing$, then $[\![\varphi]\!](S) = \varnothing$.*

Proof. We proceed by induction on the structure of φ. Let S a stream and assume φ' is a subformula of φ such that $[\![\varphi']\!](S) = \varnothing$. We assume that φ' is a proper subformula, as otherwise the result immediately follows. For this reason, we can trivially skip the case when $\varphi = R$ AS x or $\varphi = (\varphi_1)+$.

- If $\varphi = \varphi_1 ; \varphi_2$, then φ' is a subformula of φ_1 or of φ_2. Assume w.l.o.g. that φ' is a subformula of φ_1. By induction hypothesis, as $[\![\varphi']\!](S) = \varnothing$ we have that $[\![\varphi_1]\!](S) = \varnothing$, which immediately implies that $[\![\varphi]\!](S) = \varnothing$.
- If $\varphi = \varphi_1$ FILTER $P(\bar{x})$, we know that φ' is a subformula of φ_1. By induction hypothesis we have $[\![\varphi']\!](S) = \varnothing$ and by definition of FILTER we obtain $[\![\varphi]\!](S) = \varnothing$.

◂

Now we are ready to show that any core-CEL formula in disjunctive-normal form can be translated into a safe formula, and moreover, this can be done in linear time.

▸ **Lemma 11.** *Let φ be a core-CEL formula in disjunctive-normal form. Then φ can be translated in linear time into a safe core-CEL formula φ'.*

Proof. Assume that $\varphi = \varphi_1$ OR $\cdots$ OR φ_n is a core-CEL formula in disjunctive-normal form. By induction, we assume that every subformula of the form $(\varphi')+$ is already safe. Now we show that every unsafe φ_i is unsatisfiable, and therefore it can be safely removed from the disjunction. Proceed by contradiction and assume φ_i is unsafe and satisfiable. Then, it must contain a subformula of the form $\psi_1 ; \psi_2$ occurring outside the scope of all + operators, and such that $\text{vdef}_+(\psi_1) \cap \text{vdef}_+(\psi_2) \neq \varnothing$. Let $x \in \text{vdef}_+(\psi_1) \cap \text{vdef}_+(\psi_2)$. By Lemma 10, we know that $\psi_1 ; \psi_2$ must be satisfiable. Therefore, there is a stream S, a valuation ν and a mapping C such that $C \in [\![\psi_1 ; \psi_2]\!](S, \nu)$. This implies the existence of two complex events C_1 and C_2 such that $C_1 \in [\![\psi_1]\!](S, \nu)$ and $C_2 \in [\![\psi_2]\!](S, \nu)$. Since $x \in \text{vdef}_+(\psi_1)$ and ψ_1 can only mention OR inside a + operator, by Lemma 9 we obtain that $\nu(x) \in C_1$. Similarly, as $x \in \text{vdef}_+(\psi_2)$, we have $\nu(x) \in C_2$. But as $C = C_1 \cdot C_2$, we have that $C_1 \cap C_2 = \varnothing$, contradicting the facts that $\nu(x) \in C_1$ and $\nu(x) \in C_2$.

We have obtained that if any disjunct is unsafe, it cannot produce any results. Therefore, as safeness is easily verifiable, the result readily follows by removing the unsafe disjuncts of φ. Notice that this need to be done in a bottom-up fashion, starting from the subformulas of the form $(\varphi')+$. ◂

Theorem 2 occurs as a corollary of Lemmas 8 and 11. Indeed, given a core-CEL formula φ, one can construct in exponential time an equivalent core-CEL formula φ' in disjunctive

normal form. Then, from φ' one can construct in linear time a safe formula in core-CEL ψ that is equivalent to φ, which is exactly what we wanted to show. ◂

B.2 Proof of Theorem 3

Without lost of generality, in the proof we consider only unary predicates, since these are the ones that we need to modify in order for the formula to be in LP-normal form. Indeed, if the formula contains non-unary filters, this can be treated as normal operators, similar than `OR` or ; operators. Consider a well-formed core-CEL formula φ with unary predicates. We first provide a construction for a core-CEL formula in LP normal form and then prove that it is equivalent to φ. The construction consists of two steps: (1) pop predicates up, and (2) push predicates down.

1. The first step is focused on rewriting the formula in a way that for every subformula of the form φ' `FILTER` $P(x)$ it holds that $x \in \text{vdef}_+(\varphi')$. Recall that a well-formed formula could still have a subformula φ' `FILTER` $P(x)$ such that $x \notin \text{vdef}_+(\varphi')$. The construction we provide to achieve this is the following. For every subformula of the form φ' `FILTER` $P(x)$ and every predicate, let φ_x be the lowest subformula of φ with $x \in \text{vdef}_+(\varphi_x)$ and that has φ' as a subformula. Here we use the fact that φ is well-formed, which ensures that φ_x must exist. Then, we rewrite the subformula φ_x inside φ as φ_x^t `FILTER` $P(x)$ `OR` φ_x^f `FILTER` $\neg P(x)$, where φ_x^t and φ_x^f are the same as φ_x but replacing the inside $P(x)$ with `TRUE` and `FALSE`, respectively.
2. Now that we moved each predicate up to a level where all its variables are defined, the next step is to move each one down to its variable's definition. This is done straightforward: for every subformula of the form φ' `FILTER` $P(x)$, the $P(x)$ filter is removed from φ' and instead applied over every subformula of φ' with the form R `AS` x, rewriting it as R `AS` x `FILTER` $P(x)$. After all predicates were moved to the lowest possible, each assignment R `AS` x now has a sequence of filters applied to it, e.g. R `AS` x `FILTER` $P_1(x) \ldots$ `FILTER` $P_k(x)$, and moreover, all filters appear in this form. Because the predicate set $\mathbf{P}$ is closed under intersection, we know there is some $P \in \mathbf{P}$ that equals $P_1 \cap \ldots \cap P_k$. Then, we replace each sequence of filters R `AS` x `FILTER` $P_1(x) \ldots$ `FILTER` $P_k(x)$ with R `AS` x `FILTER` $P(x)$, thus resulting in a formula in LP-normal form.

Now we prove that the construction above satisfies the lemma, i.e., $[\![\varphi_{lp}]\!](S) = [\![\varphi]\!](S)$ for every stream S, where φ_{lp} is the resulting formula after the construction. To prove that the first step does not change the semantics, we show that it stays the same after each iteration. Consider a subformula φ' `FILTER` $P(x)$ of φ such that $x \notin \text{vdef}_+(\varphi')$. In particular, the only part of φ that is modified by the algorithm is φ_x, so it suffices to prove that $C \in [\![\varphi_x]\!](S, \nu)$ holds iff $C \in [\![\varphi_x^t$ `FILTER` $P(x)$ `OR` φ_x^f `FILTER` $\neg P(x)]\!](S, \nu)$.

- For the only-if direction, let S, C, ν be any stream, complex event and valuation, respectively, such that $C \in [\![\varphi_x]\!](S, \nu)$. If $S[\nu(x)] \in P$, then it is enough to prove that $C \in [\![\varphi_x^t]\!](S, \nu)$. In a similar way, the only part in which φ_x^t differs with φ_x is that in the former the atom $P(x)$ was set to `TRUE`. Therefore, it is enough to prove that, for any C' and φ', if $S[\nu'(x)] \in P$ holds, then $C' \in [\![\varphi'$ `FILTER` $P(x)]\!](S, \nu')$ iff $C' \in [\![\varphi'$ `FILTER TRUE`$]\!](S, \nu')$, which is trivially true. Notice that we can assure $S[\nu'(x)] \in P$ holds because $S[\nu(x)] \in P$ holds and, when evaluating this part of the formula, the mapping for x must stay the same, otherwise x must have been inside a +-operator, which cannot be the case because $x \in \text{bound}(\varphi_x)$. Moreover, ν' has to be

equal to ν. The proof for the case $S[\nu(x_1)] \in \neg P$ is analogous considering φ_x^f instead of φ_x^t, thus $C \in [\![\varphi_x^t \texttt{ FILTER } P(x) \texttt{ OR } \varphi_x^f \texttt{ FILTER } \neg P(x)]\!](S,\nu)$.

- For the if direction, let S, C, ν be some arbitrary stream, complex event and valuation, respectively, such that $C \in [\![\varphi_x^t \texttt{ FILTER } P(x) \texttt{ OR } \varphi_x^f \texttt{ FILTER } \neg P(x)]\!](S,\nu)$. Then, by definition, C is either in $[\![\varphi_x^t \texttt{ FILTER } P(x)]\!](S,\nu)$ or in $[\![\varphi_x^f \texttt{ FILTER } \neg P(x)]\!](S,\nu)$. Without loss of generality, consider the former case, which implies that $S[\nu(x)] \in P$. Then, because $C' \in [\![\varphi' \texttt{ FILTER } P(x)]\!](S,\nu')$ iff $C' \in [\![\varphi' \texttt{ FILTER TRUE}]\!](S,\nu')$, it holds that $C \in [\![\varphi_x]\!](S,\nu)$. It is the same for $S[\nu(x)] \in \neg P$, thus $C \in [\![\varphi_x]\!](S,\nu)$ iff $C \in [\![\varphi_x^t \texttt{ FILTER } P(x) \texttt{ OR } \varphi_x^f \texttt{ FILTER } \neg P(x)]\!](S,\nu)$.

Therefore, if we name φ_1 as the result of applying the first part, we get that $\varphi_1 \equiv \varphi$.

Now, we prove that moving the predicates to their definitions does not affect the semantics either, for which we show that it stays the same after each iteration. Consider a subformula of φ_1 of the form $\varphi' \texttt{ FILTER } P(x)$. The same way as before, we focus on the modified part, i.e., we need to prove that $C \in [\![\varphi' \texttt{ FILTER } P(x)]\!](S,\nu)$ iff $C \in [\![\varphi'_P]\!](S,\nu)$, where φ'_P is the result of adding the filter $P(x)$ for each definition of x inside φ', i.e., replace $R \texttt{ AS } x$ with $R \texttt{ AS } x \texttt{ FILTER } P(x)$ where R is any relation.

- First, we show the only-if direction. Let S, C, ν be any stream, complex event and valuation, respectively, such that $C \in [\![\varphi' \texttt{ FILTER } P(x)]\!](S,\nu)$, which implies that $S[\nu(x)] \in P$. We know that, when evaluating every subformula $R \texttt{ AS } x$ of φ', the valuation ν must stay the same, because $x \in \text{bound}(\varphi')$, and thus its definition cannot be inside a +-operator (notice that if it appears inside a +, it represents a value different to x, thus the + subformula can be rewritten using a new variable x'). Similar to the reasoning above, it holds that for any C' and φ', if $S[\nu'(x)] \in P$, then $C' \in [\![R \texttt{ AS } x \texttt{ FILTER } P(x)]\!](S,\nu')$ iff $C' \in [\![R \texttt{ AS } x]\!](S,\nu')$. Then, every subformula $R \texttt{ AS } x$ behaves the same, thus $C \in [\![\varphi'_P]\!](S,\nu)$ holds.
- We now show the if direction. Let S, C, ν be any stream, complex event and valuation, respectively, such that $C \in [\![\varphi'_P]\!](S,\nu)$. We prove that $S[\nu(x)] \in P$ must hold, thus proving that $C \in [\![\varphi' \texttt{ FILTER } P(x)]\!](S,\nu)$ holds using the same argument as above. By contradiction, assume that $S[\nu(x)] \notin P$. Because we showed that when evaluating every $R \texttt{ AS } x \texttt{ FILTER } P(x)$ in φ'_P, the valuation ν must be the same, the only possible way for $C \in [\![\varphi'_P]\!](S,\nu)$ to hold is if all $R \texttt{ AS } x$ appear at one side of an OR -operator. However, this would contradict the fact that $x \in \text{bound}(\varphi')$, thus $S[\nu(x)] \in P$, and also $C \in [\![\varphi' \texttt{ FILTER } P(x)]\!](S,\nu)$.

Then, $\varphi' \texttt{ FILTER } P(x)$ and φ'_P are equivalent, therefore, if φ_{lp} is the result of applying step 2, we get that $\varphi_{lp} \equiv \varphi_1 \equiv \varphi$.

Finally, it is easy to check that the size of φ_{lp} will be at most exponential in the size of φ. Each iteration of step 1 could duplicate the size of the formula in the worst case, thus $|\varphi_1| = \mathcal{O}(2^{|\varphi|})$. Then, step 2 does not really increase the size of the formula due to the final replacing of predicates $P_1, \ldots, P_k$ with P. The size of φ_{lp} w.r.t. φ is then $\mathcal{O}(2^{|\varphi|})$. However, in our framework (Section 3) we assumed that φ did not use the syntactic sugar $\wedge$ and $\vee$ inside its filters. If so, we argue that this would not turn into an extra exponential growth (turning the result to a double-exponential). To explain why, consider that φ uses the $\wedge$ and $\vee$ syntactic sugar. Then, if we apply step 1 to each predicate, the resulting formula φ_1 would still be equivalent to φ and of size at most exponential w.r.t. $|\varphi|$, avoiding the double-exponential blow-up mentioned above. Finally, we have that $|\varphi_{lp}| = \mathcal{O}(2^{|\varphi|})$, even if φ uses $\wedge$ and $\vee$. ◂

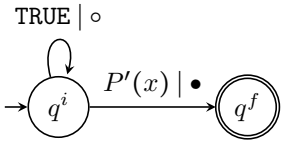
C Proofs of Section 6

C.1 Proof of Theorem 4

So simplify the proof, we will add to the model of CEA the ability to have ϵ-transitions. Formally, now a transition relation has the structure $\Delta \subseteq Q \times ((\mathbf{U} \times \{\bullet, \circ\}) \cup \{\epsilon\}) \times Q$. This basically means the automaton can have transitions of the form (p, ϵ, q) that can be part of a run and, if so, the automaton passes from state p to q without reading nor marking any new tuple. This does not give any additional power to CEA, since any ϵ-transition (p, ϵ, q) can be removed by adding, for each incoming transition of p, an equivalent incoming one to q, and for each outgoing transition of q an equivalent outgoing one from p.

The results of Theorem 3 and Theorem 2 show that we can rewrite every core-CEL formula as a safe formula in LP-normal form. We consider that, if φ is not in LP-normal form, then it is first turned into one that is, adding an exponential growth from the beginning. Furthermore, if it is not safe the it is turned into a safe one, adding another exponential growth. We now give a construction that, for every safe core-CEL formula φ in LP-normal form, defines an equivalent CEA $\mathcal{A}$, i.e., that for every complex event C, $C \in [\![\mathcal{A}]\!](S)$ iff $C \in [\![\varphi]\!](S)$. This construction is done recursively in a bottom-up fashion such that, for every subformula, an equivalent CEA is built from the CEA of its subformulas. Moreover, we assume that the CEA for each subformula has one initial state and one final state, since each recursive construction defines a CEA with those properties. Let ψ be a subformula of φ. Then, the CEA $\mathcal{A}$ is defined as follows:

- If $\psi = R$ `AS` x `FILTER` $P(x)$ then $\mathcal{A} = (Q, \Delta, \{q^i\}, \{q^f\})$ with the set of states $Q = \{q^i, q^f\}$ and the transitions $\Delta = \{(q^i, (\texttt{TRUE}, \circ), q^i), (q^i, (P', \bullet), q^f)\}$, where $P'(x) = (\text{type}(x) = R) \wedge P(x)$. Graphically, the automaton is:



  If ψ has no `FILTER` the automaton is the same but with $P'(x) = (\text{type}(x) = R)$.
- If $\psi = \psi_1$ `OR` ψ_2, and $\mathcal{A}_1 = (Q_1, \Delta_1, \{q_1^i\}, \{q_1^f\})$ and $\mathcal{A}_2 = (Q_2, \Delta_2, \{q_2^i\}, \{q_2^f\})$ are the CEA for ψ_1 and ψ_2, respectively, then $\mathcal{A} = (Q, \Delta, \{q^i\}, \{q^f\})$ where Q is the union of the states of $\mathcal{A}_1$ and $\mathcal{A}_2$ plus the new initial and final states q^i, q^f, and Δ is the union of Δ_1 and Δ_2 plus the empty transitions from q^i to the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$, and from the final states of $\mathcal{A}_1$ and $\mathcal{A}_2$ to q^f. Formally, $Q = Q_1 \cup Q_2 \cup \{q^i, q^f\}$ and $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q^i, \epsilon, q_1^i), (q^i, \epsilon, q_2^i), (q_1^f, \epsilon, q^f), (q_2^f, \epsilon, q^f)\}$.
- If $\psi = \psi_1 \,;\, \psi_2$, consider that $\mathcal{A}_1 = (Q_1, \Delta_1, \{q_1^i\}, \{q_1^f\})$ and $\mathcal{A}_2 = (Q_2, \Delta_2, \{q_2^i\}, \{q_2^f\})$ are the CEA for ψ_1 and ψ_2, respectively. Then, we define $\mathcal{A} = (Q, \Delta, \{q_1^i\}, \{q_2^f\})$, where the set of states is $Q = Q_1 \cup Q_2$ and the transition relation is $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_1^f, \epsilon, q_2^i)\}$.
- If $\psi = \psi_1+$, consider that $\mathcal{A}_1 = (Q_1, \Delta_1, \{q_1^i\}, \{q_1^f\})$ is the automaton for ψ_1. Then, we define $\mathcal{A} = (Q_1, \Delta, \{q_1^i\}, \{q_1^f\})$ where $\Delta = \Delta_1 \cup \{(q_1^f, \epsilon, q_1^i)\}$. Basically, is the same automaton for ψ_1 with an ϵ-transition from the final to the initial state.

Now, we need to prove that the previous construction satisfies Theorem 4. We will prove this by induction over the subformulas of φ, i.e., assume as induction hypothesis that the theorem holds for any subformula ψ and its respective CEA $\mathcal{A}$.

First, consider the base case $\psi = R$ AS x FILTER $P(x)$. If $C \in [\![\mathcal{A}]\!](S)$ then there is a run ρ that gets to the accepting state such that events$(\rho) = C$. Moreover, ρ must pass through the transition $(q^i, (\text{type}(x) = R \wedge P(x), \bullet), q^f)$ while reading a tuple t_j at some position j. Then, consider a valuation ν such that $\nu(x) = j$. Clearly, $C = \{\nu(x)\}$, type$(t_j) = R$, and $S[\nu(x)] \in P$, thus $C \in [\![\psi]\!](S, \nu)$. For the other direction, consider that $C \in [\![\psi]\!](S, \nu)$ for some valuation ν. Then C must contain only one position $j = \nu(x)$ such that type$(S[j]) = R$ and $S[j] \in P$ hold. Then $\rho = (q^i, (\texttt{TRUE}, \circ), q^i)^j \cdot (q^i, (P'(x), \bullet), q^f)$ is an accepting run of $\mathcal{A}$ over S, where $(q^i, (\texttt{TRUE}, \circ), q^i)^j$ means that it takes the initial loop transition j times. Because events$(\rho) = \{j\} = C$, then $C \in [\![\mathcal{A}]\!](S)$.

Now, consider the case $\psi = \psi_1$ OR ψ_2. If $C \in [\![\mathcal{A}]\!](S)$, then there is an accepting run ρ that also represents either an accepting run of $\mathcal{A}_1$ or $\mathcal{A}_2$ (removing the ϵ transitions at the beginning and end). Assume w.l.o.g. that it is the former case. Then, by induction hypothesis, there is a valuation ν such that $C \in [\![\psi_1]\!](S, \nu)$. By definition this means that $C \in [\![\psi]\!](S, \nu)$. For the other direction, consider that $C \in [\![\psi]\!](S, \nu)$ for some valuation ν. Then, either $C \in [\![\psi_1]\!](S, \nu)$ or $C \in [\![\psi_2]\!](S, \nu)$ holds. Without loss of generality, consider the former case. By induction hypothesis, it means that $C \in [\![\mathcal{A}_1]\!](S)$, so there is an accepting run ρ' of $\mathcal{A}_1$ over S such that events$(\rho') = C$. Because Δ contains Δ_1 then the run $\rho = (q^i, \epsilon, q_1^i) \cdot \rho' \cdot (q_1^f, \epsilon, q^f)$ is an accepting run of $\mathcal{A}$ over S.

Next, consider the case $\psi = \psi_1 \,;\, \psi_2$. If $C \in [\![\mathcal{A}]\!](S)$, then there is an accepting run ρ of the form $\rho : \rho_1 \cdot (q_1^f, \epsilon, q_2^i) \cdot \rho_2$ and, because of the construction, $C_1 = \text{events}(\rho_1) \in [\![\mathcal{A}_1]\!](S)$ and $C_2 = \text{events}(\rho_2) \in [\![\mathcal{A}_2]\!](S_j)$, with $j = \max(C_1) + 1$. Then by induction hypothesis there are valuations ν_1 and ν_2 such that $C_1 \in [\![\psi_1]\!](S, \nu_1)$, $C_2 \in [\![\psi_2]\!](S, \nu_2)$. Moreover, because φ is safe, we know that $\text{vdef}_+(\psi_1) \cap \text{vdef}_+(\psi_2) = \varnothing$. Therefore, we can define ν such that $\nu(x) = \nu_1(x)$ if $x \in \text{vdef}_+(\psi_1)$ and $\nu(x) = \nu_2(x)$ if $x \in \text{vdef}_+(\psi_2)$. Clearly, because ν represents both ν_1 and ν_2, it holds that $C \in [\![\psi]\!](S, \nu)$. For the other direction, consider a complex event C such that $C \in [\![\psi]\!](S, \nu)$ for some valuation ν. Then there exist complex events C_1 and C_2 such that $C_1 \in [\![\psi_1]\!](S, \nu)$, $C_2 \in [\![\psi_2]\!](S, \nu)$ and $C = C_1 \cdot C_2$. By induction hypothesis, there exist an accepting run ρ_1 of $\mathcal{A}_1$ over S such that events$(\rho_1) = C_1$. Similarly, there exist an accepting run ρ_2 of $\mathcal{A}_2$ over S_j with $j = \max(C_1)+1$ such that events$(\rho_2) = C_2$. Then, the run of $\mathcal{A}$ that simulates ρ_1 ends at a state q_1^f, thus it can continue by simulating ρ_2 and reaching a final state. Therefore, such run ρ is an accepting run of $\mathcal{A}$. Notice that events$(\rho) = C$, thus $C \in [\![\mathcal{A}]\!](S)$.

Finally, consider the case $\psi = \psi_1+$. If $C \in [\![\mathcal{A}]\!](S)$, it means that there is an accepting run ρ of $\mathcal{A}$ over S. We define k to be the number of times that ρ passes through the final state q^f, and prove by induction over k that $C \in [\![\psi]\!](S, \nu)$. If $k = 1$, it means that ρ is also an accepting run of $\mathcal{A}_1$, thus $C \in [\![\mathcal{A}_1]\!](S)$ and, by (the first) induction hypothesis, there exists some valuation ν such that $C \in [\![\psi_1]\!](S, \nu)$, which implies $C \in [\![\psi]\!](S, \nu)$. Now, consider the case $k > 1$. It means that ρ has the form $\rho = \rho_1 \cdot (q^f, \epsilon, q^i) \cdot \rho_2$ where ρ_2 passes through q^f $k-1$ times. Then, $C_1 = \text{events}(\rho_1)$ is an accepting run of $\mathcal{A}_1$, hence $C_1 \in [\![\psi_1]\!](S, \nu)$ for some ν. Furthermore, ρ_2 is an accepting run of $\mathcal{A}$, thus if $C_2 = \text{events}(\rho_2)$ then by induction hypothesis $C_2 \in [\![\psi]\!](S_j, \nu)$ for some ν, where $j = \max(C_1) + 1$. Since $C = C_1 \cdot C_2$ then $C \in [\![\psi_1 \,;\, \psi_1+]\!](S, \nu)$, thus $C \in [\![\psi]\!](S, \nu)$. Note that we do not care about ν because the +-operator of ψ overwrites it. For the other direction, consider a complex event C such that $C \in [\![\psi]\!](S, \nu)$ for some valuation ν. Then there exists ν' such that either $C \in [\![\psi_1]\!](S, \nu[\nu' \to U])$ or $C \in [\![\psi_1 \,;\, \psi_1+]\!](S, \nu[\nu' \to U])$ where $U = \text{vdef}_+(\psi_1)$. We now prove, by induction over the number of iterations, that $C \in [\![\mathcal{A}]\!](S)$. If there is just one iteration, then $C \in [\![\psi_1]\!](S, \nu[\nu' \to U])$ and, by induction hypothesis, $C \in [\![\mathcal{A}_1]\!](S)$, so there is an accepting run ρ of $\mathcal{A}_1$ over S such that events$(\rho) = C$. Because $\Delta_1 \subseteq \Delta$, then ρ is also

an accepting run of $\mathcal{A}$, thus $C \in [\![\mathcal{A}]\!](S)$. If there are k iterations with $k > 1$, it means that $C \in [\![\psi_1 \,;\, \psi_1+]\!](S, \nu[\nu' \to U])$. Therefore, there exist complex events C_1 and C_2 such that $C = C_1 \cdot C_2$, $C_1 \in [\![\psi_1]\!](S, \nu[\nu' \to U])$ and $C_2 \in [\![\psi_1+]\!](S_j, \nu[\nu' \to U])$, where $j = \max(C_1) + 1$. Then, by induction hypothesis, there exist accepting runs ρ_1 of $\mathcal{A}_1$ over S and ρ_2 of $\mathcal{A}$ over S_j such that $\text{events}(\rho_1) = C_1$ and $\text{events}(\rho_2) = C_2$ and, because $\Delta_1 \subseteq \Delta$, ρ_1 is also an accepting run of $\mathcal{A}$. Then, the run $\rho = \rho_1 \cdot (q^f, \epsilon, q^i) \cdot \rho_2$ is an accepting run of $\mathcal{A}$ over S. Furthermore, $\text{events}(\rho) = C_1 \cdot C_2 = C$ thus $C \in [\![\mathcal{A}]\!](S)$.

Finally, it is clear that the size of $\mathcal{A}$ is linear with respect to the size of φ if φ is already safe and in LP-normal form. As stated at the beginning, if φ is not safe and/or in LP-normal form, it first has to be turned into an equivalent ψ that is, and such that $|\psi| = \mathcal{O}(\exp^2(|\varphi|))$ in the worst-case scenario, where $\exp(x) = 2^x$. Then, $|\mathcal{A}| = \mathcal{O}(|\varphi|)$ if φ is safe and in LP-normal form, and $|\mathcal{A}| = \mathcal{O}(|\psi|) = \mathcal{O}(\exp^2(|\varphi|))$ otherwise. ◂

C.2 Proof of Theorem 5

C.2.1 `STRICT` operator

Consider a CEA $\mathcal{A} = (Q, \Delta, I, F)$. We will first define the CEA $\mathcal{A}_{\texttt{STRICT}}$ as $\mathcal{A}_{\texttt{STRICT}} = (Q_{\texttt{STRICT}}, \Delta_{\texttt{STRICT}}, I_{\texttt{STRICT}}, F_{\texttt{STRICT}})$ and then prove that it is equivalent to $\texttt{STRICT}(\mathcal{A})$. The set of states is defined as $Q_{\texttt{STRICT}} = \{q^m \mid q \in Q \text{ and } m \in \{\bullet, \circ\}\}$, the transition relation is $\Delta_{\texttt{STRICT}} = \{(p^m, (P, m), q^m) \mid (p, (P, m), q) \in \Delta)\} \cup \{(p^\circ, (P, \bullet), q^\bullet) \mid (p, (P, \bullet), q) \in \Delta\}$, the initial states are $I_{\texttt{STRICT}} = \{q^\circ \mid q \in I\}$ and the final states are $F_{\texttt{STRICT}} = \{q^\bullet \mid q \in F\}$. Basically, there are two copies of $\mathcal{A}$, the first one which only have the $\circ$ transitions, and the second one which only have the $\bullet$ ones, and at any $\bullet$ transition it can move from the first on to the second. On an execution, $\mathcal{A}_{\texttt{STRICT}}$ starts in the first copy of $\mathcal{A}$, moving only through transitions that do not mark the positions, until it decides to mark one. At that point it moves to the second copy of $\mathcal{A}$, and from there on it moves only using transitions with $\bullet$ until it reaches an accepting state.

Now, we prove that the construction is correct, that is, $[\![\mathcal{A}_{\texttt{STRICT}}]\!](S) = [\![\texttt{STRICT}(\mathcal{A})]\!](S)$ for every S. Let S be any stream. First, consider a complex event $C \in [\![\texttt{STRICT}(\mathcal{A})]\!](S)$. This means that $C \in [\![\mathcal{A}]\!](S)$ and that C has the form $C = \{m_1, \ldots, m_k\}$ with $m_i = m_{i-1} + 1$. Therefore, there is an accepting run of $\mathcal{A}$ of the form:

$$\rho : q_0 \xrightarrow{P_0/\circ} q_1 \xrightarrow{P_1/\circ} \cdots \xrightarrow{P_{m_1-1}/\circ} q_{m_1} \xrightarrow{P_{m_1}/\bullet} q_{m_2} \xrightarrow{P_{m_2}/\bullet} \cdots \xrightarrow{P_{m_k}/\bullet} q_{m_k+1}$$

Such that $\text{events}(\rho) = C$. Consider now the run over $\mathcal{A}_{\texttt{STRICT}}$ of the form:

$$\rho' : q_0^\circ \xrightarrow{P_0/\circ} q_1^\circ \xrightarrow{P_1/\circ} \cdots \xrightarrow{P_{m_1-1}/\circ} q_{m_1}^\circ \xrightarrow{P_{m_1}/\bullet} q_{m_2}^\bullet \xrightarrow{P_{m_2}/\bullet} \cdots \xrightarrow{P_{m_k}/\bullet} q_{m_k+1}^\bullet$$

It is clear that all transitions of ρ' are in $\Delta_{\texttt{STRICT}}$, because the ones with $\circ$ are in the first copy of $\mathcal{A}$, the first one with $\bullet$ passes from the first copy to the second, and the following ones with $\bullet$ are in the second copy. Therefore ρ' is indeed run of $\mathcal{A}_{\texttt{STRICT}}$ over S, and because $q_{m_k} \in F$, then $q_{m_k}^\bullet \in F$ and ρ' is an accepting run. Moreover, $\text{events}(\rho') = C$, thus $C \in [\![A_{\texttt{STRICT}}]\!](S)$.

Now, consider a complex event $C \in [\![A_{\texttt{STRICT}}]\!](S)$, of the form $C = \{m_1, \ldots, m_k\}$. It means that there is an accepting run of $\mathcal{A}_{\texttt{STRICT}}$ of the form:

$$\rho : q_0^\circ \xrightarrow{P_1/\circ} \cdots \xrightarrow{P_{m_1-1}/\circ} q_{m_1}^\circ \xrightarrow{P_{m_1}/\bullet} q_{m_2}^\bullet \xrightarrow{P_{m_2}/\bullet} \cdots \xrightarrow{P_{m_k}/\bullet} q_{m_k+1}^\bullet$$

Such that $\text{events}(\rho) = C$. Notice that ρ must have this form because of the structure of $\mathcal{A}_{\texttt{STRICT}}$, which force ρ to have $\circ$ transitions at the beginning and $\bullet$ ones at the end. Consider then the run of $\mathcal{A}$ of the form:

$$\rho' : q_0 \xrightarrow{P_1/\circ} \cdots \xrightarrow{P_{m_1-1}/\circ} q_{m_1} \xrightarrow{P_{m_1}/\bullet} q_{m_2} \xrightarrow{P_{m_2}/\bullet} \cdots \xrightarrow{P_{m_k}/\bullet} q_{m_k+1}$$

Similar to the converse case, it is clear that all transitions in ρ' are in Δ. Therefore ρ' is an accepting run of $\mathcal{A}$ over S, and because $\text{events}(\rho') = C$, it holds that $C \in [\![\texttt{STRICT}(\mathcal{A})]\!](S)$.

Finally, notice that $\mathcal{A}_{\texttt{STRICT}}$ consists in duplicating $\mathcal{A}$, thus the size of $\mathcal{A}_{\texttt{STRICT}}$ is two times the size of $\mathcal{A}$. ◂

C.2.2 NXT operator

Let $\mathcal{R}$ be a schema and $\mathcal{A} = (Q, \Delta, I, F)$ be a CEA over $\mathcal{R}$. In order to define the new CEA $\mathcal{A}_{\texttt{NXT}} = (Q_{\texttt{NXT}}, \Delta_{\texttt{NXT}}, I_{\texttt{NXT}}, F_{\texttt{NXT}})$ we first need to introduce some notation. We begin by imposing an arbitrary linear order $<$ between the states of Q, i.e., for every two different states $p, q \in Q$, either $p < q$ or $q < p$. Let $T_1 \dots T_k$ be a sequence of sets of states such that $T_i \subseteq Q$. We say that a sequence $T_1 \dots T_k$ is a *total preorder* over Q if $T_i \cap T_j = \varnothing$ for every $i \neq j$. Notice that the sequence is not necessarily a partition, i.e., it does not need to include all states of Q. A total preorder naturally defines a preorder between states where "p is less than q" whenever $p \in T_i$, $q \in T_j$, and $i < j$. To simplify notation, we define the concatenation between set of states such that $T \cdot T' = TT'$ whenever T and T' are non-empty and $T \cdot T' = T \cup T'$ otherwise. The concatenation between sets will help to remove empty sets during the final construction. Now, given any sequence $T_1 \dots T_k$ (not necessarily a total preorder), one can convert $T_1 \dots T_k$ into a total preorder by applying the operation *Total Pre-Ordering* (TPO) defined as follows:

$$\text{TPO}(T_1 \dots T_k) = U_1 \cdot \ldots \cdot U_k \quad \text{where} \quad U_i = T_i - \bigcup_{j=1}^{i-1} T_j.$$

Let $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ be the set of all predicates in the transitions of Δ. Define the equivalence relation $=_{\mathcal{P}}$ between tuples such that, for every pair of tuples t_1 and t_2, $t_1 =_{\mathcal{P}} t_2$ holds if, and only if, both satisfy the same predicates, i.e., $t_1 \in P_i$ holds iff $t_2 \in P_i$ holds, for every i. Moreover, for every tuple t let $[t]_{\mathcal{P}}$ represent the equivalence class of t defined by $=_{\mathcal{P}}$, that is, $[t]_{\mathcal{P}} = \{t' \mid t =_{\mathcal{P}} t'\}$. Notice that, even though there are infinitely many tuples, there is a finite amount of equivalence classes which is bounded by all possible combinations of predicates in $\mathcal{P}$, i.e., $2^{|\mathcal{P}|}$. Now, for every $S \subseteq \{1, \dots, n\}$, define the predicate:

$$P_S = \left(\bigwedge_{i \in S} P_i\right) \wedge \left(\bigwedge_{i \notin S} \neg P_i\right)$$

and define the new set of predicates $\mathcal{P}$-types $= \{P_S \mid S \subseteq \{1, \dots, n\}\}$. Notice that for every tuple t there is exactly one predicate in $\mathcal{P}$-types that is satisfied by t; we call that predicate P_t. Finally, we extend the transition relation Δ as a function such that:

$$\Delta(T, P, m) = \{q \in Q \mid \text{exist } p \in T \text{ and } P' \in \mathcal{P} \text{ such that } P \subseteq P' \text{ and } (p, (P', m), q) \in \Delta\}$$

for every $T \subseteq Q$, $P \in \mathcal{P}$-types, and $m \in \{\bullet, \circ\}$.

In the sequel, we define the CEA $\mathcal{A}_{\texttt{NXT}} = (Q_{\texttt{NXT}}, \Delta_{\texttt{NXT}}, I_{\texttt{NXT}}, F_{\texttt{NXT}})$ component by component. First, the set of states $Q_{\texttt{NXT}}$ is defined as

$$Q_{\texttt{NXT}} = \{(T_1 \dots T_k, p) \mid T_1 \dots T_k \text{ is a total preorder over } Q \text{ and } p \in T_i \text{ for some } i \leq k\}$$

Intuitively, the state p is the current state of the 'simulation' of $\mathcal{A}$ and the sets $T_1 \dots T_k$ contain the states in which the automaton could be, considering the prefix of the word read until the current moment. Furthermore, the sets are ordered consistently with respect to $\leq_{\text{next}}$, e.g., if a run ρ_1 reach the state $(\{1,2\}\{3\}, 1)$ and other run ρ_2 reach the state $(\{1,2\}\{3\}, 3)$, then $\text{events}(\rho_2) <_{\text{next}} \text{events}(\rho_1)$. This property is proven later in Lemma 12.

Secondly, the transition relation is defined as follows. Consider $P \in \mathcal{P}$-types, $m \in \{\bullet, \circ\}$ and $(\mathcal{T}, p), (\mathcal{U}, q) \in Q_{\texttt{NXT}}$ where $\mathcal{T} = T_1 \dots T_k$ and $p \in T_i$ for some $i \leq k$. Then we have that $((\mathcal{T}, p), P, m, (\mathcal{U}, q)) \in \Delta_{\texttt{NXT}}$ if, and only if,

1. $(p, P', m, q) \in \Delta$ for some P' such that $P \subseteq P'$,
2. $q \notin \Delta(T_j, P, m')$ for every $m' \in \{\bullet, \circ\}$ and $j < i$,
3. $\mathcal{U} = \text{TPO}(U_1^\bullet \cdot U_1^\circ \cdot \ldots \cdot U_k^\bullet \cdot U_k^\circ)$ where $U_j^\bullet = \Delta(T_j, P, \bullet)$ and $U_j^\circ = \Delta(T_j, P, \circ)$ for $1 \leq j \leq k$,
4. $q \notin \Delta(T_i, P, \bullet)$ when $m = \circ$, and
5. $(p', P', m, q) \notin \Delta$ for every $p' \in T_i$ such that $p' < p$ and every P' such that $P \subseteq P'$.

Intuitively, the first condition ensures that the 'simulation' respects the transitions of Δ, the second checks that the next state could not have been reached from a 'higher' run, the third ensures that the sequence is updated correctly and the fourth restricts that if the next state can be reached either marking the letter or not, it always choose to mark it. The last condition is not strictly necessary, and removing it will not change the semantics of the automaton, but is useful because it ensures that there are no two runs ρ_1 and ρ_2 that end in the same state such that $\text{events}(\rho_1) = \text{events}(\rho_2)$.

Finally, the initial set $I_{\texttt{NXT}}$ is defined as all states of the form (I, q) where $q \in I$ and the final set $F_{\texttt{NXT}}$ as all states of the form $(T_1 \dots T_k, p)$ such that $p \in F$ and there exists $i \leq k$ such that $p \in T_i$ and $T_j \cap F = \varnothing$ for all $j < i$.

Let $S = t_1 t_2 \dots$ be any stream. To prove that the construction is correct, we will need the following lemma.

▸ **Lemma 12.** *Consider a CEA $\mathcal{A} = (Q, \Delta, I, F)$, a stream S, two states $(\mathcal{T}, p), (\mathcal{T}, q) \in Q_{\texttt{NXT}}$ with the same sequence $\mathcal{T} = T_1 \dots T_k$ such that $p \in T_i$, $q \in T_j$ for some i and j, and two runs ρ_1, ρ_2 of $\mathcal{A}_{\texttt{NXT}}$ over S that have the same length and reach the states $(\mathcal{T}, p)$ and $(\mathcal{T}, q)$, respectively. Then, $i < j$ if, and only if:*

$$\text{events}(\rho_2) <_{\text{next}} \text{events}(\rho_1)$$

Proof. We will prove it by induction over the length of the runs. Let $q_0, q_0' \in I$ be any two initial states of $\mathcal{A}$, not necessarily different. First, assume that both runs consist of reading a single tuple t. Then, the runs are of the form:

$$\rho_1 : (I, q_0) \xrightarrow{P_t/m_1} (\mathcal{T}, p) \qquad \text{and} \qquad \rho_2 : (I, q_0') \xrightarrow{P_t/m_2} (\mathcal{T}, q)$$

where $\mathcal{T} = T_1 T_2 = \text{TPO}(\Delta(I, P_t, \bullet)\Delta(I, P_t, \circ))$ and neither T_1 nor T_2 can be empty because p and q are in different sets. For the if direction, the only option is that $\text{events}(\rho_1) = \{1\}$ and $\text{events}(\rho_2) = \{\}$, which implies that $m_1 = \bullet$ and $m_2 = \circ$. Then $i < j$ because $p \in T_1$ and $q \in T_2$. For the only-if direction, because $i < j$ then $p \in T_1$ and $q \in T_2$, so necessarily $m_1 = \bullet$ and $m_2 = \circ$. Because of this, $\text{events}(\rho_1) = \{1\}$ and $\text{events}(\rho_2) = \{\}$, therefore $\text{events}(\rho_2) <_{\text{next}} \text{events}(\rho_1)$. Now, let $S = t_1 t_2 \dots t_n \dots$ and consider that the runs are of the form:

$$\rho_1 : (I, q_0) \xrightarrow{P_{t_1}/m_1} (\mathcal{T}_1, q_1) \xrightarrow{P_{t_2}/m_2} \dots \xrightarrow{P_{t_{n-1}}/m_{n-1}} (\mathcal{T}_{n-1}, q_{n-1}) \xrightarrow{P_{t_n}/m_n} (\mathcal{T}, p)$$
$$\rho_2 : (I, q_0') \xrightarrow{P_{t_1}/m_1'} (\mathcal{T}_1, q_1') \xrightarrow{P_{t_2}/m_2'} \dots \xrightarrow{P_{t_{n-1}}/m_{n-1}'} (\mathcal{T}_{n-1}, q_{n-1}') \xrightarrow{P_{t_n}/m_n'} (\mathcal{T}, q)$$

Notice that both runs have the same sequences $\mathcal{T}_1, \dots, \mathcal{T}_{n-1}$ because each sequence $\mathcal{T}_i$ is defined only by the previous sequence $\mathcal{T}_{i-1}$ and the tuple t_i which implicitly defines the predicate P_{t_i}. Furthermore, all the runs over the same word must have the same sequences. Define the runs ρ_1' and ρ_2', respectively, as the runs ρ_1 and ρ_2 without the last transition.

Consider that $\mathcal{T}_{n-1}$ has the form $\mathcal{T}_{n-1} = U_1U_2\dots U_k$, and that $q_{n-1} \in U_r$ and $q'_{n-1} \in U_s$ for some r and s. Notice that, because of the construction, if it is the case that $r < s$ ($r > s$), then $i < j$ ($i > j$ resp.) must hold. For the if direction, consider that events$(\rho_2) <_{\text{next}}$ events(ρ_1). If events(ρ'_1) = events(ρ'_2), by induction hypothesis it means that $r = s$. Moreover, the only option is that $m_n = \bullet$ and $m'_n = \circ$, therefore, by the construction it holds that $i < j$. If events$(\rho'_2) <_{\text{next}}$ events(ρ'_1), by induction hypothesis it means that $r < s$ and because of the construction, $i < j$. Notice that events$(\rho'_1) <_{\text{next}}$ events(ρ'_2) cannot occur because the lower element of events(ρ'_2) not in events(ρ'_1) would still be the lower element of events(ρ_2) not in events(ρ_1), thus contradicting events$(\rho_2) <_{\text{next}}$ events(ρ_1). For the only-if direction, consider that $i < j$. It is easy to see that, if $r > s$, then i cannot be lower than j, thus we do not consider this case. Now, consider the case that $r = s$. Because $i < j$, it must occur that $m_n = \bullet$ and $m'_n = \circ$, so events(ρ_1) = events$(\rho'_1) \cup \{n\}$ and events(ρ_2) = events(ρ'_2). By induction hypothesis, events(ρ'_1) = events(ρ'_2), therefore events$(\rho_2) <_{\text{next}}$ events(ρ_1). Consider now the case that $r < s$. By induction hypothesis, events$(\rho'_2) <_{\text{next}}$ events(ρ'_1) and, because the last transition can only add n to both complex events, it follows that events$(\rho_2) <_{\text{next}}$ events(ρ_1). ◂

Now, we need to prove that if $C \in [\![\texttt{NXT}(\mathcal{A})]\!](S)$, then $C \in [\![\mathcal{A}_{\texttt{NXT}}]\!](S)$ and vice versa. First, consider a complex event $C \in [\![\mathcal{A}_{\texttt{NXT}}]\!](S)$. To prove that $C \in [\![\texttt{NXT}(\mathcal{A})]\!](S)$, we need to show that $C \in [\![\mathcal{A}]\!](S)$ and that for all complex events C' such that $C \leq_{\texttt{NXT}} C'$ and $\max(C) = \max(C')$, $C' \notin [\![\mathcal{A}]\!](S)$. Assume that the run associated to C is:

$$\rho : (\mathcal{U}_0, q_0) \xrightarrow{P_{t_1}/m_1} (\mathcal{U}_1, q_1) \xrightarrow{P_{t_2}/m_2} \dots \xrightarrow{P_{t_n}/m_n} (\mathcal{U}_n, q_n)$$

Because of the construction of Δ (in particular, the first condition), for every i it holds that $(q_{i-1}, P_i, m_i, q_i) \in \Delta$ for some P_i such that $P_{t_i} \subseteq P_i$. Because $t_i \in P_{t_i}$, then $t_i \in P_i$, thus the run:

$$\rho' : q_0 \xrightarrow{P_1/m_1} q_1 \xrightarrow{P_2/m_2} \dots \xrightarrow{P_n/m_n} q_n$$

is an accepting run of $\mathcal{A}$ over S, and thus $C \in [\![\mathcal{A}]\!](S)$. Now, recall from construction of $F_{\texttt{NXT}}$ that there exists $i \leq k$ such that $q_n \in T_i$ and $T_j \cap F = \varnothing$ for all $j < i$, where $T_1 \dots T_k = \mathcal{U}_n$. Then, because of Lemma 12, $C' <_{\text{next}} C$ for every other $C' \in [\![\mathcal{A}]\!](S)$ such that $\max(C) = \max(C')$, otherwise the run of C' would end in an accepting state inside a T_j such that $j < i$, which cannot happen. Therefore, $C \in [\![\texttt{NXT}(\mathcal{A})]\!](S)$.

Now, consider a complex event $C \in [\![\texttt{NXT}(\mathcal{A})]\!](S)$. Assume that the run associated to C is:

$$\rho : q_0 \xrightarrow{P_1/m_1} q_1 \xrightarrow{P_2/m_2} \dots \xrightarrow{P_n/m_n} q_n$$

To prove that $C \in [\![\mathcal{A}_{\texttt{NXT}}]\!](S)$ we will prove that there exists an accepting run on $\mathcal{A}_{\texttt{NXT}}$. Based on ρ, consider now the run:

$$\rho' : (\mathcal{U}_0, p_0) \xrightarrow{P_{t_1}/m_1} (\mathcal{U}_1, p_1) \xrightarrow{P_{t_2}/m_2} \dots \xrightarrow{P_{t_n}/m_n} (\mathcal{U}_n, p_n)$$

Where the complex events $m_1, \dots, m_n$ are the same, each condition P_{t_i} is defined by t_i and each $\mathcal{U}_i$ is the result of applying the function TPO based on $\mathcal{U}_{i-1}$ and P_{t_i} Moreover, each p_i is defined as follows. As notation, consider that $\mathcal{U}_i = T^i_1 \dots T^i_{k_i}$ and that every q_i is in the r_i-th set of $\mathcal{U}_i$, i.e., $q_i \in T^i_{r_i}$. Then, p_i is the lower state in $T^i_{r_i}$ such that $(p_i, (P_{t_{i+1}}, m_{i+1}), p_{i+1}) \in \Delta$, and $p_n = q_n$. Notice that ρ' is completely defined by ρ and S. We will prove that ρ' is an accepting run by checking that all transitions meet the conditions of the transition relation $\Delta_{\texttt{NXT}}$. Now, it is clear that the first condition is satisfied by all transitions, i.e., for every i it holds that $(p_{i-1}, (P', m_i), p_i) \in \Delta$ for some P' such that $P_{t_i} \subseteq P'$ (just consider $P' = P_i$). For

the second condition, by contradiction suppose that it is not satisfied by ρ'. It means that for some i, $p_i \in \Delta(T_j^{i-1}, P_i, m')$ for some $m' \in \{\bullet, \circ\}$ and $j < r_i$. In particular, consider that the state $p' \in T_j^{i-1}$ is the one for which $(p', P', m', p_i) \in \Delta$. Recall that every state inside a sequence is reachable considering the prefix of the word read until that moment. This means that there exist the accepting runs:

$$\sigma : q'_0 \xrightarrow{P'_1/m'_1} q'_1 \xrightarrow{P'_2/m'_2} \cdots \xrightarrow{P'_{i-1}/m'_{i-1}} q' \xrightarrow{P'_i/m'_i} q_i \xrightarrow{P_{i+1}/m_{i+1}} \cdots \xrightarrow{P_n/m_n} q_n$$

$$\sigma' : (\mathcal{U}_0, p'_0) \xrightarrow{P_{t_1}/m'_1} (\mathcal{U}_1, p'_1) \xrightarrow{P_{t_1}/m'_2} \cdots \xrightarrow{P_{t_{i-1}}/m'_{i-1}} (\mathcal{U}_{i-1}, p') \xrightarrow{P_{t_i}/m'_i} (\mathcal{U}_i, p_i) \xrightarrow{P_{t_{i+1}}/m_{i+1}} \cdots \xrightarrow{P_{t_n}/m_n} (\mathcal{U}_n, p_n)$$

Where p'_i are defined in a similar way to p_i. Define for every run γ and every i the run γ_i as γ until the i-th transition. For example, ρ_i is equal to the run ρ until the state q_i. Then, by Lemma 12, $\text{events}(\rho'_{i-1}) < \text{events}(\sigma'_{i-1})$, but $\text{events}(\rho') = \text{events}(\sigma')$. This is a contradiction, since $\text{events}(\rho')$ and $\text{events}(\sigma')$ differ from $\text{events}(\rho'_{i-1})$ and $\text{events}(\sigma'_{i-1})$ in that the formers can contain additional positions from i to n, but the minimum position remains in $\text{events}(\sigma'_{i-1})$, and therefore in $\text{events}(\sigma')$. The fourth condition is proven by contradiction too. Suppose that it is not satisfied by ρ', which means that for some i, $p_i \in \Delta(T_{r_{i-1}}^{i-1}, P_{t_i}, \bullet)$ when $m_i = \circ$. Then, the run:

$$\sigma : p_0 \xrightarrow{P_{t_1}/m_1} p_1 \xrightarrow{P_{t_2}/m_2} \cdots \xrightarrow{P_{t_{i-1}}/m_{i-1}} p_{i-1} \xrightarrow{P_{t_i}/\bullet} p_i \xrightarrow{P_{t_{i+1}}/m_{i+1}} \cdots \xrightarrow{P_{t_n}/m_n} p_n$$

is an accepting run such that $\text{events}(\rho) < \text{events}(\sigma)$, which is a contradiction, since $C \in [\![\texttt{NXT}(\mathcal{A})]\!](S)$. The third and last conditions are trivially proven because of the construction of the run. Therefore, ρ' is a valid run of $\mathcal{A}_{\texttt{NXT}}$ over S. Moreover, because $p_n = q_n \in F$ then ρ' is an accepting run, therefore $\text{events}(\rho) = C \in [\![\mathcal{A}_{\texttt{NXT}}]\!](S)$.

Now, we analyze the properties of the automaton $\mathcal{A}_{\texttt{NXT}}$. First, we show that $|\mathcal{A}_{\texttt{NXT}}|$ is at most exponential over $|\mathcal{A}|$. Notice that each state in $Q_{\texttt{NXT}}$ represents a sequence of subsets of Q, thus each state has at most $|Q|$ subsets. Moreover, for each one of the subsets there are at most $2^{|Q|}$ possible combinations. Therefore, there the resulting automaton $\mathcal{A}_{\texttt{NXT}}$ is of size exponential over $|\mathcal{A}|$. ◂

C.2.3 `LAST` operator

The `LAST` case is done in the same way as the `NXT` one, with some minor changes. We now define the CEA $\mathcal{A}_{\texttt{LAST}} = (Q_{\texttt{LAST}}, \Delta_{\texttt{LAST}}, I_{\texttt{LAST}}, F_{\texttt{LAST}})$ component by component. First, the set of states $Q_{\texttt{LAST}}$ is defined exactly like $Q_{\texttt{NXT}}$:

$$Q_{\texttt{LAST}} = \{(T_1 \dots T_k, p) \mid T_1 \dots T_k \text{ is a total preorder over } Q \text{ and } p \in T_i \text{ for some } i \leq k\}$$

The intuition in this case is that the sets will be ordered consistently with respect to $\leq_{\text{last}}$, e.g., if a run ρ_1 reach the state $(\{1,2\}\{3\}, 1)$ and other run ρ_2 reach the state $(\{1,2\}\{3\}, 3)$, then $\text{events}(\rho_2) <_{\text{last}} \text{events}(\rho_1)$. This property can be proven in the same way as Lemma 12.

Secondly, the transition relation is defined as follows. Consider $P \in \mathcal{P}$-types, $m \in \{\bullet, \circ\}$ and $(\mathcal{T}, p), (\mathcal{U}, q) \in Q_{\texttt{LAST}}$ where $\mathcal{T} = T_1 \dots T_k$ and $p \in T_i$ for some $i \leq k$. Then we have that $((\mathcal{T}, p), P, m, (\mathcal{U}, q)) \in \Delta_{\texttt{LAST}}$ if, and only if,

1. $(p, P', m, q) \in \Delta$ for some P' such that $P \subseteq P'$,
2. $q \notin \Delta(T_j, P, m')$ for every $m' \in \{\bullet, \circ\}$ and $j < i$,
3. $\mathcal{U} = \text{TPO}(U_1^\bullet \cdot \ldots \cdot U_k^\bullet \cdot U_1^\circ \cdot \ldots \cdot U_k^\circ)$ where $U_j^\bullet = \Delta(T_j, P, \bullet)$ and $U_j^\circ = \Delta(T_j, P, \circ)$ for $1 \leq j \leq k$,

4. $q \notin \Delta(T_i, P, \bullet)$ when $m = \circ$, and
5. $(p', P', m, q) \notin \Delta$ for every $p' \in T_i$ such that $p' < p$ and every P' such that $P \subseteq P'$.

The only condition that changes with respect to the NXT case is the third one. In this case, it ensures that the sequence is updated correctly according to the last-order. In particular, it changes the order $U_1^\bullet \cdot U_1^\circ \cdot \ldots \cdot U_k^\bullet \cdot U_k^\circ$ in the NXT case with $U_1^\bullet \cdot \ldots \cdot U_k^\bullet \cdot U_1^\circ \cdot \ldots \cdot U_k^\circ$. Intuitively, in the first one, if the run reads event at position i and adds it to its complex event C (turning into $C \cup \{i\}$), then it "wins" over the case in which that same run did not mark it. On the other hand, in the second case, if the run reads event at position i and adds it to its complex event C (turning into $C \cup \{i\}$), then it "wins" over every other run that did not mark it.

Finally, the initial and final sets are defined like for the NXT: $I_{\texttt{LAST}}$ is defined as all states of the form (I, q) where $q \in I$ and $F_{\texttt{LAST}}$ as all states of the form $(T_1 \ldots T_k, p)$ such that $p \in F$ and there exists $i \leq k$ such that $p \in T_i$ and $T_j \cap F = \varnothing$ for all $j < i$.

The proof of correctness is a direct replicate of the one for the NXT case, changing only the notation from NXT to LAST.

C.2.4 MAX operator

Let $\mathcal{A} = (Q, \Delta, I, F)$ be a CEA. Similarly to the construction of CEA for the NXT, we define the set $\mathcal{P}$-types such that for every tuple t there is exactly one predicate P_t in $\mathcal{P}$-types that is satisfied by t, and extend the transition relation Δ as a function $\Delta(T, P, m)$ for every $T \subseteq Q$, $P \in \mathcal{P}$-types, and $m \in \{\bullet, \circ\}$. Further, we overload the notation of Δ as a function such that $\Delta(T, P) = \Delta(T, P, \bullet) \cup \Delta(T, P, \circ)$.

We now define the CEA $\mathcal{A}_{\texttt{MAX}} = (Q_{\texttt{MAX}}, \Delta_{\texttt{MAX}}, I_{\texttt{MAX}}, F_{\texttt{MAX}})$ component by component. First, the set of states is $Q_{\texttt{MAX}} = \{(S, T) \mid S, T \subseteq Q,\ S \neq \varnothing \text{ and } S \cap T = \varnothing\}$. At each $(S, T) \in Q_{\texttt{MAX}}$, S will keep track of the states of $\mathcal{A}$ that are reached by runs that define the same complex event C (and are not in T), and T will keep track of the states that are reached by runs that define a complex event C' such that $C \subset C'$. The transition relation is $\Delta_{\texttt{MAX}} = \Delta^\bullet_{\texttt{MAX}} \cup \Delta^\circ_{\texttt{MAX}}$, with

$$\Delta^\bullet_{\texttt{MAX}} = \{((S_1, T_1), (P, \bullet), (S_2, T_2)) \mid T_2 = \Delta(T_1, P, \bullet) \ \wedge \ S_2 = \Delta(S_1, P, \bullet) \setminus T_2\}$$
$$\Delta^\circ_{\texttt{MAX}} = \{((S_1, T_1), (P, \circ), (S_2, T_2)) \mid T_2 = \Delta(T_1, P) \cup \Delta(S_1, P, \bullet) \wedge S_2 = \Delta(S_1, P, \circ) \setminus T_2\}$$

The former updates T_1 to T_2 using $\bullet$-transitions from T_1, and S_1 to S_2 the same way but removing the ones from T_2. The latter updates T_1 to T_2 using all transitions from T_1 plus the $\bullet$-transitions from S_1, while it updates S_1 to S_2 using $\circ$-transitions from S_1. Finally, $I_{\texttt{MAX}} = \{(I, \varnothing)\}$, and $F_{\texttt{MAX}} = \{(S, T) \in Q_{\texttt{MAX}} \mid S \cap F \neq \varnothing \text{ and } T \cap F = \varnothing\}$.

Next, we prove the above, i.e., $C \in [\![\texttt{MAX}(\mathcal{A})]\!](S)$ iff $C \in [\![\mathcal{A}_{\texttt{MAX}}]\!](S)$. To simplify the proof, we assume that $\mathcal{A}$ is I/O-deterministic, therefore each state of $Q_{\texttt{MAX}}$ now has the form (q, T). The proof can easily be extended for non I/O-deterministic $\mathcal{A}$. First, we prove the if direction. Consider a complex event C such that $C \in [\![\mathcal{A}_{\texttt{MAX}}]\!](S)$. To prove that $C \in [\![\texttt{MAX}(\mathcal{A})]\!](S)$, we first prove that $C \in [\![\mathcal{A}]\!](S)$ by giving an accepting run of $\mathcal{A}$ associated to C. Assume that the run of $\mathcal{A}_{\texttt{MAX}}$ over S associated to C is:

$$\rho : (q_0, T_0) \xrightarrow{P_{t_1}/m_1} (q_1, T_1) \xrightarrow{P_{t_2}/m_2} \ldots \xrightarrow{P_{t_n}/m_n} (q_n, T_n)$$

Where $T_0 = \varnothing$, $T_n \cap F = \varnothing$ and $((q_{i-1}, T_{i-1}), (P_{t_i}, m_i), (q_i, T_i)) \in \Delta_{\texttt{MAX}}$. Furthermore, $q_0 \in I$ and $q_n \in F$. Also, from the construction of $\Delta_{\texttt{MAX}}$, we deduce that for every i there is a predicate P_i such that $(q_{i-1}, (P_i, m_i), q_i) \in \Delta$. This means that the run:

$$q_0 \xrightarrow{P_1/m_1} q_1 \xrightarrow{P_2/m_2} \ldots \xrightarrow{P_n/m_n} q_n$$

Is an accepting run of $\mathcal{A}$ associated to C. Now, we prove by contradiction that for every C' such that $C \subset C'$, $C' \notin [\![\mathcal{A}]\!](S)$. In order to do this, we define the next lemma, in which we use the notion of *partial run*, which is the same as a run but not necessarily beginning at an initial state.

▸ **Lemma 13.** *Consider an I/O-deterministic CEA $\mathcal{A} = (Q, \Delta, I, F)$, a stream $S = t_1 t_2 \dots$ and two partial runs of $\mathcal{A}_{\texttt{MAX}}$ and $\mathcal{A}$ over S, respectivelly:*

$$\sigma : (q_0, T_0) \xrightarrow{P_{t_1}/m_1} (q_1, T_1) \xrightarrow{P_{t_2}/m_2} \cdots \xrightarrow{P_{t_n}/m_n} (q_n, T_n)$$
$$\sigma' : p_0 \xrightarrow{P_1/m'_1} p_1 \xrightarrow{P_2/m'_2} \cdots \xrightarrow{P_n/m'_n} p_n$$

Then, if $p_0 \in T_0$ and $m'_i = \bullet$ at every i for which $m_i = \bullet$, it holds that $p_n \in T_n$.

Proof. This is proved by induction over the length n. First, if $n = 0$, then $p_n = p_0$ and $T_n = T_0$, so $p_n \in T_n$. Now, assume that the lemma holds for $n-1$, i.e., $p_{n-1} \in T_{n-1}$. Consider the case that $m_n = \bullet$. Then $m'_n = \bullet$ also, thus $(p_{n-1}, (P_n, \bullet), p_n) \in \Delta$. Furthermore, $T_n = \Delta(T_{n-1}, P_{t_n}, \bullet)$ and therefore $p_n \in T_n$, because $p_{n-1} \in T_{n-1}$. Now, consider the case $m_n = \circ$. Either $(p_{n-1}, (P_n, \bullet), p_n) \in \Delta$ or $(p_{n-1}, (P_n, \circ), p_n) \in \Delta$, so $p_n \in \Delta(T_{n-1}, P_{t_n})$. Moreover, $\Delta(T_{n-1}, P_{t_n}) \subseteq T_n$ because of the construction of $\Delta_{\texttt{MAX}}$, therefore $p_n \in T_n$. ◂

Now, by contradiction consider a complex event C' such that $C \subset C'$ and $C' \in [\![\mathcal{A}]\!](S)$. Then, there must exist an accepting run of $\mathcal{A}$ over S associated to C' of the form:

$$\rho' : p_0 \xrightarrow{P'_1/m'_1} p_1 \xrightarrow{P'_2/m'_2} \cdots \xrightarrow{P'_n/m'_n} p_n$$

such that $m'_i = \bullet$ at every i for which $m_i = \bullet$, and there is at least one i for which $m_i = \circ$ and $m'_i = \bullet$. Consider i to be the lower position for which this happens. Because $\mathcal{A}$ is I/O-deterministic, ρ' can be rewritten as:

$$\rho' : q_0 \xrightarrow{P_1/m_1} \cdots \xrightarrow{P_{i-1}/m_{i-1}} q_{i-1} \xrightarrow{P'_i/\bullet} p_i \xrightarrow{P'_{i+1}/m'_{i+1}} \cdots \xrightarrow{P'_n/m'_n} p_n$$

Similarly, to ease visualization we rewrite ρ as:

$$\rho : (q_0, T_0) \xrightarrow{P_{t_1}/m_1} \cdots \xrightarrow{P_{t_{i-1}}/m_{i-1}} (q_{i-1}, T_{i-1}) \xrightarrow{P_{t_i}/\circ} (q_i, T_i) \xrightarrow{P_{t_{i+1}}/m_{i+1}} \cdots \xrightarrow{P_{t_n}/m_n} (q_n, T_n)$$

In particular, the transition $((q_{i-1}, T_{i-1}), (P_{t_i}, \circ), (q_i, T_i))$ is in $\Delta_{\texttt{MAX}}$, which means that $\Delta(\{q_{i-1}\}, P_{t_i}, \bullet) \subseteq T_i$. Moreover, $(q_{i-1}, (P'_i, \bullet), p_i) \in \Delta$ and, because $t_i \in P'_i$, then $P_{t_i} \subseteq P'_i$ thus $p_i \in T_i$. Now, by Lemma 13 it follows that $p_n \in T_n$. But because ρ is an accepting run, we get that $T_n \cap F = \varnothing$ and so $p_n \notin F$, which is a contradiction to the statement that ρ' is an accepting run. Therefore, for every C' such that $C \subset C'$, $C' \notin [\![\mathcal{A}]\!](S)$, hence $C \in [\![\texttt{MAX}(\mathcal{A})]\!](S)$.

Next, we will prove the only-if direction. For this, we will need the following lemma:

▸ **Lemma 14.** *Consider an I/O-deterministic CEA $\mathcal{A} = (Q, \Delta, I, F)$, a stream $S = t_1 t_2 \dots$, a run of $\mathcal{A}_{\texttt{MAX}}$ over S:*

$$\sigma : (q_0, T_0) \xrightarrow{P_{t_1}/m_1} (q_1, T_1) \xrightarrow{P_{t_2}/m_2} \cdots \xrightarrow{P_{n_2}/m_n} (q_n, T_n)$$

And a state $p \in Q$. If $p \in T_n$, then there is a run of $\mathcal{A}$ over S:

$$\sigma' : p_0 \xrightarrow{P_1/m'_1} p_1 \xrightarrow{P_2/m'_2} \cdots \xrightarrow{P_{n-1}/m'_{n-1}} p_{n-1} \xrightarrow{P_n/m'_n} p$$

Such that $\text{events}(\sigma) \subset \text{events}(\sigma')$.

Proof. It will be proved by induction over the length n. The base case is $n = 0$, which is trivially true because $T_0 = \varnothing$. Assume now that the Lemma holds for $n-1$. Define the run σ_{n-1} as the run σ without the last transition. For any state $q \in T_{n-1}$, let σ'_q be the run that ends in q such that $\text{events}(\sigma_{n-1}) \subset \text{events}(\sigma'_q)$. Consider the case $m_n = \circ$. Then, either $p \in \Delta(T_{n-1}, P_{t_n})$ or $p \in \Delta(\{q_{n-1}\}, P_{t_n}, \bullet)$. In the former scenario, there must be a $q \in T_{n-1}$ and $P \in \mathbf{U}$ such that $(q, (P_n, m), p) \in \Delta$ and $P_{t_n} \subseteq P$, with $m \in \{\bullet, \circ\}$. Define σ' as the run σ'_q followed by the transition $(q, (P, m), p)$. Then σ' satisfies $\text{events}(\sigma) \subset \text{events}(\sigma')$. In the latter scenario, there must be an $P \in \mathbf{U}$ such that $(q_{n-1}, (P, \bullet), p) \in \Delta$ and $P_{t_n} \subseteq P$. Define σ' as σ_{n-1} followed by the transition $(q_{n-1}, (P, \bullet), p)$. Then σ' satisfies $\text{events}(\sigma) \subset \text{events}(\sigma')$. Now, consider the case $m_n = \bullet$. Here, p has to be in $\Delta(T_{n-1}, P_{t_n}, \bullet)$, so there must be a $q \in T_{n-1}$ and $P \in \mathbf{U}$ such that $(q, (P, \bullet), p) \in \Delta$ and $P_{t_n} \subseteq P$. Define σ' as the run σ'_q followed by the transition $(q, (P, \bullet), p)$. Then σ' satisfies $\text{events}(\sigma) \subset \text{events}(\sigma')$. Finally, the Lemma holds for every n. ◂

Consider a complex event C such that $C \in [\![\texttt{MAX}(\mathcal{A})]\!](S)$. This means that there is an accepting run of $\mathcal{A}$ over S associated to C. Define that run as:

$$\rho : q_0 \xrightarrow{P_1/m_1} q_1 \xrightarrow{P_2/m_2} \dots \xrightarrow{P_n/m_n} q_n$$

Where $q_0 \in I$, $q_n \in F$ and $(q_{i-1}, (P_i, m_i), q_i) \in \Delta$. To prove that $C \in [\![\mathcal{A}_{\texttt{MAX}}]\!](S)$ we give an accepting run of $\mathcal{A}_{\texttt{MAX}}$ over S associated to C. Consider the run:

$$\rho' : (q_0, T_0) \xrightarrow{P_{t_1}/m_1} (q_1, T_1) \xrightarrow{P_{t_2}/m_2} \dots \xrightarrow{P_{t_n}/m_n} (q_n, T_n)$$

Where $T_0 = \varnothing$, $T_i = \Delta(T_{i-1}, P_{t_i}) \cup \Delta(\{q_{i-1}\}, P_{t_i}, \bullet)$ if $m_i = \circ$, and $T_i = \Delta(T_{i-1}, P_{t_i}, \bullet)$ if $m_i = \bullet$. To be a valid run, every transition $(T_{i-1}, P_{t_i}, m_i, T_i)$ must be in $\Delta_{\texttt{MAX}}$, which we prove now by induction over i. The base case is $i = 0$, which is trivially true because no transition is required to exist. Next, assume that transitions up to $i-1$ exist. We know that there is a P_i such that $P_{t_i} \subseteq P_i$ and $(q_{i-1}, (P_i, m_i), q_i) \in \Delta$, so that condition is satisfied. We only need to prove that $q_i \notin T_i$. By contradiction, assume that $q_i \in T_i$. Consider the case that $m_i = \circ$. It means that either $q_i \in \Delta(\{q_{i-1}\}, P_{t_i}, \bullet)$ or $q_i \in \Delta(T_{i-1}, P_{t_i})$. In the first scenario, consider a new run σ to be exactly the same as ρ, but changing m_i with $\bullet$. Then σ is also an accepting run, and $\text{events}(\rho) \subset \text{events}(\sigma)$, which is a contradiction to the definition of the MAX semantic. In the second scenario, there must be some $p \in T_{i-1}$ and $P \in \mathbf{U}$ such that $(p, (P, m), q_i) \in \Delta$, where $m \in \{\bullet, \circ\}$. Because of Lemma 14, it means that there is a run σ' over S:

$$\sigma' : p_0 \xrightarrow{P'_1/m'_1} p_1 \xrightarrow{P'_2/m'_2} \dots \xrightarrow{P'_{i-2}/m'_{i-2}} p_{i-2} \xrightarrow{P'_{i-1}/m'_{i-1}} p$$

Such that $\text{events}(\rho_{i-1}) \subset \text{events}(\sigma')$, where ρ_{i-1} is the run ρ until transition $i-1$. Moreover, because $(p, (P, m), q_i) \in \Delta$ we can define the run:

$$\sigma : p_0 \xrightarrow{P'_1/m'_1} p_1 \xrightarrow{P'_2/m'_2} \dots \xrightarrow{P'_{i-1}/m'_{i-1}} p \xrightarrow{P/m} q_i \xrightarrow{P_{i+1}/m_{i+1}} \dots \xrightarrow{P_n/m_n} q_n$$

Such that $\text{events}(\rho) \subset \text{events}(\sigma)$, which is also a contradiction. Then, $q_i \notin T_i$ for the case $m_i = \circ$. Now, consider the case $m_i = \bullet$. Assuming that $q_i \in T_i$, it means that $q_i \in \Delta(T_{i-1}, P_{t_i}, \bullet)$. Then, there must be some $p \in T_{i-1}$ and $P \in \mathbf{U}$ such that $(p, (P, \bullet), q_i) \in \Delta$. Alike the previous case, because of Lemma 14, there is a run:

$$\sigma : p_0 \xrightarrow{P'_1/m'_1} p_1 \xrightarrow{P'_2/m'_2} \dots \xrightarrow{P'_{i-1}/m'_{i-1}} p \xrightarrow{P/\bullet} q_i \xrightarrow{P_{i+1}/m_{i+1}} \dots \xrightarrow{P_n/m_n} q_n$$

Such that $\text{events}(\rho) \subset \text{events}(\sigma)$, which is a contradiction. Then, $q_i \notin T_i$, thus it holds that $(T_{i-1}, (P_{t_i}, m_i), T_i) \in \Delta_{\texttt{MAX}}$ for every i. The above proved that ρ' is a run of $\mathcal{A}_{\texttt{MAX}}$, but to be a

accepting run it must hold that $T_n \cap F = \varnothing$. By contradiction, assume otherwise, i.e., there is some $q \in Q$ such that $q \in T_n \cap F$. Then, because of Lemma 14, there is another accepting run σ of $\mathcal{A}_{\texttt{MAX}}$ over S such that $C \subset \text{events}(\sigma)$, which contradicts the fact that C is maximal. Thus, $T_n \cap F = \varnothing$ and ρ' is an accepting run, therefore $C \in [\![\mathcal{A}_{\texttt{MAX}}]\!](S)$. Finally, note that $\mathcal{A}_{\texttt{MAX}}$ is of size exponential in the size of $\mathcal{A}$, even when $\mathcal{A}$ is not I/O-deterministic. ◂

C.3 Proof of Proposition 1

We prove CEA closure under I/O-determinization. For the following proof consider an arbitrary CEA $\mathcal{A} = (Q, \Delta, I, F)$. We define the CEA $\mathcal{A}_d = (Q_d, \delta_d, I_d, F_d)$ component by component. First, the set of states is $Q_d = 2^Q$, that is, each state in Q_d represents a different subset of Q. Second, the transition relation is:

$$\delta_d = \{(T, (P, m), U) \mid P \in \mathcal{P}\text{-types, and } q \in U \text{ iff there are } p \in T \text{ and } P' \in \mathbf{U} \text{ such that } (p, (P', m), q) \in \Delta \text{ and } P \subseteq P'\}.$$

Here, $\mathcal{P}$ is the set of all predicates in the transitions of Δ and we use the notion of $\mathcal{P}$-types defined in the proof of Theorem 5 (see Section C.2.2 for the definition). Finally, the sets of initial and final states are $I_d = \{I\}$ and $F_d = \{T \in Q_d \mid T \cap F \neq \varnothing\}$. The key notion here is the one of $\mathcal{P}$-types, which partitions the set of all tuples in a way that if a tuple t satisfies a predicate $P_t \in \mathcal{P}$-types, then P_t is a subset of the predicates of all transition that a run of $\mathcal{A}$ could take when reading t. This allows us to then apply a determinization algorithm similar to the one for FSA. Notice that $P_1 \cap P_2 = \varnothing$ for every two different predicates $P_1, P_2 \in \mathcal{P}$-types, so the resulting CEA $\mathcal{A}_d$ is I/O-deterministic. ◂

D Proofs of Section 7

D.1 Proof of Theorem 6

To prove the theorem, we first show how to evaluate an I/O-deterministic CEA. Then we extend the evaluation algorithm to evaluate any CEA by doing a determinization of the automaton "on the fly".

D.1.1 Evaluation of I/O-deterministic CEA

We describe a CEP evaluation algorithm with $f(n) = n$ update time for I/O-deterministic CEA. We define the algorithm's underlying data structure, then show how to update this data structure upon new events, and finally how to enumerate the resulting complex events with constant delay.

Data structure. The atomic element in our data structure is the *node*. A node is defined as a pair (p, l), where $p \in \mathbb{N}$ represents a position in the stream and l is a list of nodes. A node is initialized by calling Node(p, l), and the methods `position` and `list` return p and l, respectively.

The data structure maintained by our algorithm is composed by linked-lists of nodes. For operating a linked-list l we use the methods `add`, `append` and `lazycopy`. Specifically, `add`(n) adds the node n at the beginning of l, and `append`(l') appends a list l' at the end of l. An important property of the data structure is that no element is ever removed from the lists, only adding nodes or appending lists is allowed. This allows us to represent a list as a pair $l = (s, e)$, where s is its starting node and e its ending node. Then, `lazycopy` returns a

Algorithm 1 Evaluate $\mathcal{A}$ over a stream S

Require: An I/O deterministic CEA $\mathcal{A} = (Q, \delta, q_0, F)$

procedure EVALUATE(S)
  for all $q \in Q \setminus \{q_0\}$ **do**
    $\text{list}_q \leftarrow \epsilon$
  $\text{list}_{q_0} \leftarrow [\bot]$
  while $t \leftarrow \texttt{yield}_S$ **do**
    for all $q \in Q$ **do**
      $\text{list}_q^{\texttt{old}} \leftarrow \text{list}_q.\texttt{lazycopy}$, $\text{list}_q \leftarrow \epsilon$
    for all $q \in Q$ **with** $\text{list}_q^{\texttt{old}} \neq \epsilon$ **do**
      if $p^\bullet \leftarrow \delta(q, t, \bullet)$ **then**
        $\text{list}_{p^\bullet}.\texttt{add}(\text{Node}(t.\texttt{position}, \text{list}_q^{\texttt{old}}))$
      if $p^\circ \leftarrow \delta(q, t, \circ)$ **then**
        $\text{list}_{p^\circ}.\texttt{append}(\text{list}_q^{\texttt{old}})$
    ENUMERATE($\{\text{list}_q\}_{q \in Q}, F, t.\texttt{position}$)

Algorithm 2 Enumerate all complex events

procedure ENUMERATE($\{\text{list}_q\}_{q \in Q}$, F, now)
  for all $q \in F$ **with** $\text{list}_q \neq \epsilon$ **do**
    $\text{list}_q.\texttt{begin}$
    while $n \leftarrow \text{list}.\texttt{next}$ **and** $n.\texttt{position} = now$ **do**
      ENUMALL($n.\texttt{list}, \{n.\texttt{position}\}$)
procedure ENUMALL(list, C)
  list.`begin`
  while $n \leftarrow \text{list}.\texttt{next}$ **do**
    if $n = \bot$ **then**
      `Output`(C)
    else
      ENUMALL($n.\texttt{list}, C \cup \{n.\texttt{position}\}$)

copy of l, defined by the pointers (s, e), and the generated *copy* of the list is not affected by future changes on l. Furthermore, it is trivial to see that `lazycopy` runs in constant time (i.e. $\mathcal{O}(1)$). The methods used for navigating the list are `begin` and `next`. `begin` gives a pointer to the first node of the list, and `next` returns the next element of the list and `false` when it reaches the end.

Evaluation. The CEP evaluation algorithm for an I/O-deterministic CEA $\mathcal{A} = (Q, \delta, q_0, F)$ is given in Algorithms 1 and 2. To ease the notation, we extend δ as a function $\delta(q, t, m)$ that retrieves the (unique) state $p = \delta(q, P, m)$ for some predicate P such that $t \in P$; if there is no such P, it returns `false`. Basically, if a run is in state q, then p is the state it moves when reading t and marking m.

The procedure EVALUATE keeps the evaluation of $\mathcal{A}$ by simulating all its possible runs, and has a list list_q for each state q to keep track on the complex events. Intuitively, each list_q keeps the information of the partial complex events generated by the partial runs currently ending at q. Each node n in list_q represents (through its $n.\texttt{list}$) a subset of these complex events, all of them having $n.\texttt{position}$ as their last position. These sets are pairwise disjoint

(which is an important property for constant-delay enumeration of the output). Each list_q is initialized as the empty list, represented by ϵ, except for list_{q_0}, which begins with only the sink node $\perp$ in it. The algorithm then reads S using $\texttt{yield}_S$ to get each new event. For each new event t, the procedure updates the data structure as follows. It starts by creating a copy of each list_q, and storing it in $\text{list}_q^{\texttt{old}}$ (lines 6-7). Then, for each q with non-empty list_q it extends the runs that are currently at q by simulating the possible outgoing transitions satisfied by t (lines 8-12). After doing this for all q, it calls the ENUMERATE procedure to enumerate all output complex events generated by t.

The core processing of Algorithm 1 is in updating the structure by extending the runs currently at q (lines 9-12). Specifically, line 10 considers the $\bullet$-transition and line 12 the $\circ$-transition (recall that $\mathcal{A}$ is I/O-deterministic). As we said before, each list_q represents the complex events of runs currently at q. To extend these runs with a $\bullet$-transition, line 10 creates a new node n^* with the current position in S (i.e. $t.\texttt{position}$) as its position, and the old value of list_q as its predecessors list. Then, n^* is added at the top of the new list of $p^\bullet = \delta(q, t, \bullet)$. On the other hand, to extend the runs with a $\circ$-transition, it only needs to append the old list of q to the list of $p^\circ = \delta(q, t, \circ)$ (line 12).

By looking at Algorithm 1, one can see that the update of each list_q takes time $\mathcal{O}(|t| \cdot |\delta|)$ by checking containment of t in each predicate P of the outgoing transitions, because we considered that containment $t \in P$ is checked in time $\mathcal{O}(|t|)$. Moreover, while iterating over each state (the **for** in line 8), we pass over each transition at most once, thus the time needed to iterate over all states is $\mathcal{O}(|t| \cdot |\delta|)$. This, added to the $\mathcal{O}(|Q|)$ of the lazy copying of the lists, gives us an overall $\mathcal{O}(|\mathcal{A}| \cdot |t|)$ bound on the time between each call to $\texttt{yield}_S$, satisfying condition **1.** with $f(|\mathcal{A}|) = |\mathcal{A}|$.

Enumeration. One can consider the data structure maintained by EVALUATE as a directed acyclic graph: vertices are nodes and there is an outgoing edge from node n to node n' if n' appears in $n.\texttt{list}$. By following Algorithm 1, one can easily check that the sink node $\perp$ is reachable from every node in this directed acyclic graph, namely, for any q and any node n in list_q there exists a path $n = n_1, \ldots, n_k, \perp$. Furthermore, each of this path represents a complex event $\{n_k.\texttt{position}, \ldots, n_1.\texttt{position}\}$ outputted by some run of $\mathcal{A}$ over S that ends at q.

Given the previous discussion, the ENUMERATE procedure in Algorithm 2 is straightforward: it simply traverses the directed acyclic graph in a depth-first manner, computing a complex event for each path. To ensure that all outputs are enumerated, it needs to do this for each node n in an accepting state and whose position is equal to the current position (i.e. *now*). Because new nodes are added on top, it iterates over each accepting list from the beginning, stopping whenever it finds a node with a position different from *now*.

Constant Delay. It is important to note that ENUMERATE does not satisfy condition **2.** of a CEP evaluation algorithm, namely, taking a constant delay between two outputs. The problem relies in the depth-first search traversal of the acyclic graph: there can be an unbounded number of backtracking steps, creating a delay that is not constant between outputs. To solve this, we provide the method ENUMALL* in Algorithm 3 which does the same as ENUMALL in Algorithm 2 and runs with constant delay. Moreover, the algorithm takes constant time between each output event (i.e. position), and constant time between complex events.

We start by explaining the notation in Algorithm 3. For doing a wise backtracking during the enumeration, we use an extended stack of nodes (denoted by s in the algorithm) that we call a *black-white stack*. This stack works as a traditional stack with the difference that stack elements are colored with black and white. For coloring the nodes, we provide

Algorithm 3 Algorithm equivalent to EnumAll that runs with constant delay

Require: $n \neq \perp$ and n.`list` is non-empty.

```
1:  procedure EnumAll*(n)
2:      n.list.begin
3:      s.push-black(n)
4:      while s.empty = false do
5:          n ← s.pop()
6:          if n = ⊥ then
7:              n ← s.pop-whites()
8:              Output(s)
9:          else
10:             Output(n)
11:             n′ ← n.list.next
12:             if n.list.atEnd then
13:                 s.push-white(n)
14:             else
15:                 s.push-black(n)
16:             n′.list.begin
17:             s.push-black(n′)
```

the methods `push-black`(n) and `push-white`(n) that assign the colored black and white, respectively, when the node n is push into the stack. This stack also has the traditional method `pop`() and `empty` for popping the top node and checking if the stack is empty, respectively. The colors are used when the method `pop-whites`() is called. When this method is called, the stack pops all the white nodes that are at the top of the stack. For example, if s = ❶②③❹⑤⑥⑦ is a black-white stack with node 7 the top of the stack, then when s.`pop-whites`() is called the resulting stack will be ❶②③❹, namely, all white nodes at the top of the stack are popped. Note that by keeping pointers to the previous black node, each method of a black-white stack can be run in constant time.

For printing the output, we assume a method `Output`(n) which prints the position of the node n.`position` in the user output tape. Furthermore, if $s = n_1 \dots n_{i-1} n_i$ is the current content of a black-white stack with n_i the top of the stack, we assume a method `Output`(s) that prints $\# \, n_1$.`position` $\dots$ n_{i-1}.`position` in the output (if s is empty or has one node, it does not print any symbol). Note that the output will be printed as a sequence $\bar{C}_1 \# \bar{C}_2 \# \dots \# \bar{C}_k$ where each $\bar{C}_i$ is a sequence of positions (i.e. a complex event) printed in reverse order, namely, if $C = \{i_1, \dots, i_k\}$ with $i_1 < \dots < i_k$, then $\bar{C} = i_k \dots i_1$. For lists, we assume an extra method `atEnd`, which returns true if the iterator of the list is at the end of the list. Finally, we assume that the node $\perp$ always has an empty list (i.e. we can apply $\perp$.`list.begin` but $\perp$.`list.atEnd` is always true).

The intuition behind EnumAll* is the following. As we previously stated, we will see the data structure that stores the complex events as an acyclic directed graph, where each node n has edges to the nodes of n.`list`. Both EnumAll and EnumAll* are based on the same intuition: to navigate through the graph in a depth-first-search manner and compute a complex event for each path from the root to a leaf (i.e. $\perp$). The main difference is that, while EnumAll does this with recursion and moves one node at a time, EnumAll* can move up an arbitrary number of nodes when it acknowledges that there are no more paths (i.e. complex events) at that section of the graph. This is achieved by the use of the black-

white stack and, specifically, in line 7 where the `pop-whites` method is called to backtrack an arbitrary number of nodes in constant time. This is particularly useful in cases when, for example, the graph consists of only two disjoint paths that meet at the root. In this scenario, after enumerating the complex event C_1 of the first path, EnumAll would have to go back to the root through $|C_1|$ nodes before enumerating the complex event C_2 for the second path, thus taking time $\mathcal{O}(|C_1|)$ between C_1 and C_2. On the other side, EnumAll* uses the black-white stack s to store the exact point at which it has to go back (in the example, the root node), therefore it takes constant time between each complex event output. Moreover, to print the partial complex event, EnumAll* uses the method `Output`(s) to recap the output from the current position and continue printing from there (line 8). This way, EnumAll* ensures that the time it takes in enumerating between positions or complex events is bounded by a constant.

Algorithm 4 Evaluate non-deterministic $\mathcal{A}$ over a stream S

Require: A non-deterministic CEA $\mathcal{A} = (Q, \Delta, I, F)$

1: **procedure** NDetEvaluate(S)
2: **for all** $T \in 2^Q \setminus \{I\}$ **do**
3: $\text{list}_T \leftarrow \epsilon$
4: $\text{list}_I \leftarrow [\bot]$
5: active $\leftarrow \{I\}$
6: **while** $t \leftarrow \texttt{yield}_S$ **do**
7: $\text{active}^{\texttt{old}} \leftarrow \text{active}.\texttt{copy}$, active $\leftarrow \varnothing$
8: **for all** $T \in \text{active}^{\texttt{old}}$ **do**
9: $\text{list}_T^{\texttt{old}} \leftarrow \text{list}_T.\texttt{lazycopy}$, $\text{list}_T \leftarrow \epsilon$
10: **for all** $T \in \text{active}^{\texttt{old}}$ **do**
11: $U^\bullet \leftarrow \Delta(T, t, \bullet)$
12: **if** $U^\bullet \neq \varnothing$ **then**
13: $\text{list}_{U^\bullet}.\texttt{add}(\text{Node}(t.\texttt{position}, \text{list}_T^{\texttt{old}}))$
14: active $\leftarrow$ active $\cup \{U^\bullet\}$
15: $U^\circ \leftarrow \Delta(T, t, \circ)$
16: **if** $U^\circ \neq \varnothing$ **then**
17: $\text{list}_{U^\circ}.\texttt{append}(\text{list}_T^{\texttt{old}})$
18: active $\leftarrow$ active $\cup \{U^\circ\}$
19: Enumerate$(\{\text{list}_T\}_{T \in 2^Q}, \{T \mid T \cap F \neq \varnothing\}, t.\texttt{position})$

D.1.2 Evaluation of any CEA

Here we provide Algorithm 4, an evaluation algorithm for evaluating an arbitrary CEA $\mathcal{A}$. The procedure NDetEvaluate is strongly based on Evaluate from Algorithm 2, modified to do a determinization of $\mathcal{A}$ "on the fly". It handles subsets of Q as its new states by keeping a list list_T for each subset of states T instead of the lists list_q for each state q. Moreover, it extends the transition Δ as a function $\Delta(T, t, m)$ that returns the set of all states reachable from some state in $T \subseteq Q$ after reading event t and marking with $m \in \{\bullet, \circ\}$; a similar extension to the one defined for Algorithm 1. With this modifications, the update of each list list_T is done the same way as Evaluate. To extend it considering $\bullet$-transitions when reading t, it creates a new node n^* with the current position $t.\texttt{position}$ and linked to the old list_T; then adds n^* at the top of the list $\text{list}_{U^\bullet}$ of $U^\bullet = \Delta(T, t, \bullet)$. On the other hand, to

extend it with $\circ$-transitions, it appends the old list of T to the list of $U^\circ = \Delta(T, t, \circ)$.

Further, a mild optimization is added in Algorithm 4. It utilizes a set active, which contains the sets T that have non-empty list_T, avoiding the need to iterate over all subsets of Q when it is not necessary. However, the exponential update time is still maintained for the worst-case scenario. It is worth noting that there is an alternative algorithm for evaluating $\mathcal{A}$ that consists in first determinizing $\mathcal{A}$ and then running Algorithm 1 on the resulting I/O-deterministic CEA $\mathcal{A}^{\text{det}}$. This evaluation algorithm updates in time linear to the size $|\mathcal{A}^{\text{det}}| = \mathcal{O}(2^{|\mathcal{A}|})$, resulting in the same update time as Algorithm 4.

D.2 Proof of Theorem 7

Here we provide for each selection strategy $\texttt{SEL} \in \{\texttt{NXT}, \texttt{LAST}, \texttt{STRICT}, \texttt{MAX}\}$ an evaluation algorithm for evaluating $\texttt{SEL}(\mathcal{A})$ for an arbitrary CEA $\mathcal{A}$. Each algorithm comes from combining the automata constructions of Theorem 5 and the evaluation algorithm for I/O deterministic CEA. Moreover, each algorithm uses the ENUMERATE procedure to enumerate all matchings, similar than the algorithm for I/O-deterministic CEA.

D.2.1 NXT evaluation

Algorithm 5 Evaluate $\mathcal{A}$ over a stream S with NXT semantics

Require: CEA $\mathcal{A} = (Q, \Delta, I, F)$

```
1:  procedure NextEvaluate(S)
2:      for all q ∈ Q ∖ I do
3:          list_q ← ϵ
4:      for all q ∈ I do
5:          list_q ← [⊥]
6:      O ← [I]
7:      while t ← yield_S do
8:          for all q ∈ Q do
9:              list_q^old ← list_q.lazycopy, list_q ← ϵ
10:         O^old ← O, O ← []
11:         for all A ∈ O^old do
12:             UpdateMarking(A, t, •)
13:             UpdateMarking(A, t, ∘)
14:         Enumerate({list_q}_{q∈Q}, F, t.position)
15: procedure UpdateMarking(A, t, m)
16:     B ← ∅
17:     for all q ∈ A and p ∈ Δ(q, t, m) ∖ O.set do
18:         B ← B ∪ {p}
19:         if m = • then
20:             list_p ← [Node(t.position, list_q^old)]
21:         else
22:             list_p ← list_q^old
23:     if B ≠ ∅ then
24:         O.enqueue(B)
```

An evaluation algorithm for $\texttt{NXT}(\mathcal{A})$ is given in Algorithm 5. The procedure NextEvaluate uses the same approach as the construction of the CEA $\mathcal{A}_{\texttt{NXT}}$ of Theorem 5, which simulated $\mathcal{A}$ while keeping an order of priority over the states. This order was used so that $\mathcal{A}_{\texttt{NXT}}$ could simulate a run ρ of $\mathcal{A}$ that reaches q only if there was no other simultaneous run ρ' reaching q and such that $\text{events}(\rho) \leq_{\text{next}} \text{events}(\rho')$. To mimic this behavior, Algorithm 5 keeps that order in a queue of set of states, called O. We assume that O has two methods: $\texttt{enqueue}(A)$ to add a set of states A to the queue and `set` to take the union of all set of states inside the queue. Furthermore, at each update, list_q stores at most one node, defined by the first state in the O-order that reaches q. This way, when traversing the structure in the Enumerate procedure, the result is at most one complex event for each list_q with $q \in F$, which is exactly the maximum complex event in the $\leq_{\text{next}}$ order that reaches q. This, however, could result in giving the same complex event more than once, when it is defined by different runs that end at different states of F. To avoid this issue, one can make sure that $|F| = 1$ by adding a new final state q_f to $\mathcal{A}$ and adding a transition (p, P, m, q_f) for each (p, P, m, q) that reaches some $q \in F$.

Regarding the update time of Algorithm 5, we examine the while iteration of line 7. First of all, note that the O-queue keeps disjoint set of states and, therefore, its length is bounded by the number of states in Q. Furthermore, for each set of states $A \in O$ the function UpdateMarking iterates over each state in A and each transition $\Delta(q, t, m)$. As we said, the sets $A \in O$ are disjoint which implies that each state and transition is checked at most once in UpdateMarking, namely, $|\mathcal{A}|$. By using a smart data structure to check membership in O in logarithmic time, the update time of Algorithm 5 is at most linear in the size of $\mathcal{A}$.

D.2.2 `LAST` evaluation

Algorithm 6 Evaluate $\mathcal{A}$ over a stream S with `LAST` semantics

Require: CEA $\mathcal{A} = (Q, \Delta, I, F)$

1: **procedure** LastEvaluate(S)
2: **for all** $q \in Q \setminus I$ **do**
3: $\text{list}_q \leftarrow \epsilon$
4: **for all** $q \in I$ **do**
5: $\text{list}_q \leftarrow [\bot]$
6: $O \leftarrow [I]$
7: **while** $t \leftarrow \texttt{yield}_S$ **do**
8: **for all** $q \in Q$ **do**
9: $\text{list}_q^{\texttt{old}} \leftarrow \text{list}_q.\texttt{lazycopy}$, $\text{list}_q \leftarrow \epsilon$
10: $O^{\texttt{old}} \leftarrow O$, $O \leftarrow []$
11: **for all** $A \in O^{\texttt{old}}$ **do**
12: UpdateMarking($A, t, \bullet$)
13: **for all** $A \in O^{\texttt{old}}$ **do**
14: UpdateMarking($A, t, \circ$)
15: Enumerate($\{\text{list}_q\}_{q \in Q}, F, t.\texttt{position}$)

Algorithm 6 is an evaluation algorithm for $\texttt{LAST}(\mathcal{A})$. One can se the resemblance of procedure LastEvaluate with NextEvaluate. In fact, both have the same approach:

keeping an order O defining how to update the lists. The difference is that LastEvaluate follows the order of $\mathcal{A}_{\texttt{LAST}}$ of Theorem 5, i.e. simulates a run ρ of $\mathcal{A}$ that reaches q only if there was no other simultaneous run ρ' reaching q and such that events$(\rho) \leq_{\text{last}}$ events(ρ'). To achieve this, it prioritizes the updates that add the last position: it iterates over all $\bullet$-transitions before all $\circ$-transitions, unlike NextEvaluate which iterates over each A state checking both $\bullet$-transitions and $\circ$-transitions from A at the same time. The same argument about the complexity of NextEvaluate applies to LastEvaluate, thus its update time is also $\mathcal{O}(|\mathcal{A}|)$.

D.2.3 `MAX` evaluation

Algorithm 7 Evaluate $\mathcal{A}$ over a stream S with `MAX` semantics

Require: CEA $\mathcal{A} = (Q, \Delta, I, F)$

1: **procedure** MaxEvaluate(S)
2: **for all** $r \in 2^Q \times 2^Q \setminus \{(I, \varnothing)\}$ **do**
3: $\text{list}_r \leftarrow \epsilon$
4: $\text{list}_{(I,\varnothing)} \leftarrow [\bot]$
5: active $\leftarrow \{(I, \varnothing)\}$
6: **while** $t \leftarrow \texttt{yield}_S$ **do**
7: $\text{active}^{\texttt{old}} \leftarrow \text{active}.\texttt{copy}$, active $\leftarrow \varnothing$
8: **for all** $r \in \text{active}^{\texttt{old}}$ **do**
9: $\text{list}_r^{\texttt{old}} \leftarrow \text{list}_r.\texttt{lazycopy}$, $\text{list}_r \leftarrow \epsilon$
10: **for all** $r \in \text{active}^{\texttt{old}}$ **do**
11: MoveMarking(r, t)
12: MoveNotMarking(r, t)
13: Enumerate($\{\text{list}_r\}_{r \in 2^Q \times 2^Q}, \{(T, U) \mid T \cap F \neq \varnothing \wedge U \cap F = \varnothing\}, t.\texttt{position}$)
14: **procedure** MoveMarking($(T, U), t$)
15: $U' \leftarrow \Delta(U, t, \bullet)$
16: $T' \leftarrow \Delta(T, t, \bullet) \setminus U'$
17: **if** $T' \neq \varnothing$ **then**
18: $\text{list}_{(T',U')}.\texttt{add}(\text{Node}(t.\texttt{position}, \text{list}^{\texttt{old}}_{(T,U)}))$
19: active $\leftarrow$ active $\cup \{(T', U')\}$
20: **procedure** MoveNotMarking($(T, U), t$)
21: $U' \leftarrow \Delta(U, t, \bullet) \cup \Delta(U, t, \circ) \cup \Delta(T, t, \bullet)$
22: $T' \leftarrow \Delta(T, t, \circ) \setminus U'$
23: **if** $T' \neq \varnothing$ **then**
24: $\text{list}_{(T',U')}.\texttt{append}(\text{list}^{\texttt{old}}_{(T,U)})$
25: active $\leftarrow$ active $\cup \{(T', U')\}$

The algorithm for evaluating $\texttt{MAX}(\mathcal{A})$ is Algorithm 7, which is arguably the most convoluted one so far. Here we use the extension of Δ we defined for Algorithm 4.

Procedure MaxEvaluate keeps for each pair $(T, U) \in 2^Q \times 2^Q$ a list $\text{list}_{(T,U)}$. Similar than for the algorithm for I/O deterministic CEA, here each list keeps the complex event data for a set of runs. The procedure initializes all lists as empty except for $\text{list}_{(I,\varnothing)}$, which begins with $\bot$ in it. At each update (lines 7-13), it first creates a lazycopy of each list. Then, each list is updated by procedures MoveMarking and MoveNotMarking (lines 11 and 12). MoveMarking updates the list with $\bullet$ transitions the same way as Algorithm 1, i.e. adding

a new node to the target list with the current t.`position`, and linking it with the origin list (line 18). However, it differs in that the origin and target lists are not defined by a transition, e.g. $(p, P, \bullet, q)$. Instead, the origin $\text{list}_{(T,U)}$ and target $\text{list}_{(T',U')}$ are bound by the relations (lines 15-16):

$$(*) \begin{cases} U' = \Delta(U, t, \bullet) \\ T' = \Delta(T, t, \bullet) \setminus U' \end{cases}$$

Moreover, MoveNoteMarking updates the list with $\circ$ transitions by appending the origin list to the target list (line 24), as in Algorithm 1. In this case, the origin $\text{list}_{(T,U)}$ and target $\text{list}_{(T',U')}$ are bound by relations (lines 21-22):

$$(**) \begin{cases} U' = \Delta(U, t, \bullet) \cup \Delta(U, t, \circ) \cup \Delta(T, t, \bullet) \\ T' = \Delta(T, t, \circ) \setminus U' \end{cases}$$

Both $(*)$ and $(**)$ are motivated by the standard automata determinization: to compress all the runs that define the same output in a single run ρ, keeping track of the set of current states T and updating it using the transition relation Δ. Here we also need to store the set U of states that are reached by runs that define superset complex events of the current one, i.e. the states that can be reached by some simultaneous run ρ' such that $\text{events}(\rho) \subseteq \text{events}(\rho')$. Because of $(*)$ and $(**)$, the runs represented by $\text{list}_{(T,U)}$ are the ones that end at some $q \in T$, and if there is other simultaneous run ρ' such chat $\text{events}(\rho) \subseteq \text{events}(\rho')$ then ρ' must end at some state $p \in U$. This way, in the call Enumerate at line 13, we give as final-states argument the pairs (T, U) such that T has an accepting state and U does not, which means that the runs in $\text{list}_{(T,U)}$ define complex events that are maximal.

As a basic optimization, a set active is stored which keeps the pairs (T, U) with non-empty $\text{list}_{(T,U)}$, avoiding the need to iterate over all pairs of $2^Q \times 2^Q$ when it is not necessary. Still, the complexity in the worst-case scenario remains exponential ($\mathcal{O}(4^{|\mathcal{A}|})$).

D.2.4 `STRICT` evaluation

Algorithm 8 Evaluate $\mathcal{A}$ over a stream S with `STRICT` semantics

Require: An I/O deterministic CEA $\mathcal{A} = (Q, \delta, q_0, F)$

```
1:  procedure StrictEvaluate(S)
2:      for all q ∈ Q \ {q0} do
3:          list_q ← ϵ
4:      q_init ← q0, list_{q_init} ← [⊥]
5:      while t ← yield_S do
6:          for all q ∈ Q do
7:              list_q^old ← list_q.lazycopy, list_q ← ϵ
8:          for all q ∈ Q with list_q^old ≠ ϵ do
9:              if p• ← δ(q, t, •) then
10:                 list_p•.add(Node(t.position, list_q^old))
11:         if q_init ← δ(q_init, t, ∘) then
12:             list_{q_init}.append([⊥])
13:         Enumerate({list_q}_{q∈Q}, F, t.position)
```

An evaluation algorithm for `STRICT`($\mathcal{A}$) is given in Algorithm 8. First, it requires $\mathcal{A}$

to be deterministic, so for evaluating an arbitrary CEA it first needs to be determinized, incurring in an additional $2^{|\mathcal{A}|}$ blow-up.

Procedure StrictEvaluate is very similar to Evaluate of Algorithm 1. The core difference is that it keeps track of a special q_{init} state that represents (if it exists) the run done by following only $\circ$ transitions, i.e. the empty run that still have not marked any position. At each update, it follows the same idea as Evaluate to update the lists considering $\bullet$ transitions. On the other hand, it does not update the lists in the same way for $\circ$. This is because it only computes the runs that define continuous intervals of events, therefore no $\circ$ transition can be taken if some event was marked with a $\bullet$ transition. Therefore, it only considers the $\circ$ transitions in the empty run: it updates q_{init} with $\delta(q_{init}, t, \circ)$ (line 11) and adds the $\perp$ node to the new $list_{q_{init}}$ (line 12). The call of Enumerate is the same as in Evaluate.

As mentioned above, we first need to determinize $\mathcal{A}$ before running Algorithm 8, which results in a $2^{|\mathcal{A}|}$ blow-up on the size of the complex event automaton. Moreover, since the algorithm runs in linear time over the input CEA (by the same arguments as Algorithm 1), the overall update time is $\mathcal{O}(2^{|\mathcal{A}|})$.