A Formal Framework for Complex Event Processing

Alejandro Grez
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile
ajgrez@uc.cl

Cristian Riveros
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile
cristian.riveros@uc.cl

Martín Ugarte
Millennium Institute for Foundational Research on Data, Santiago, Chile
martin@martinugarte.com

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.

2012 ACM Subject Classification Information systems → Data streams; Theory of computation → Data structures and algorithms for data management; Theory of computation → Database query languages (principles); Theory of computation → Automata extensions

Keywords and phrases Complex event processing, streaming evaluation, constant delay enumeration

Digital Object Identifier 10.4230/LIPIcs.ICDT.2019.5

Acknowledgements Cristian and Alejandro have been funded by FONDECYT grant 11150653, and together with Martín they were partially supported by the Millennium Institute for Foundational Research on Data (IMFD). M. Ugarte also acknowledges support from the Brussels Captial Region – Innoviris (project SPICES). We also thank the anonymous referees for their helpful comments.

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 translations 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 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 journal version.

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

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.

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 hard to extend. 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 core 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 more involved 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 $S = t_0 t_1 \ldots$ 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 $S = t_0 t_1 \ldots$ and a position $i \in \mathbb{N}$, the i-th element of S is denoted by $S[i] = t_i$, and the sub-stream $t_i t_{i+1} \ldots$ of S is denoted by S_i. Note that we consider that the time of each event is given by its index, and defer a more elaborated 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 $\varphi \;\texttt{FILTER}\; (P(\bar{x}) \wedge Q(\bar{y}))$ or $\varphi \;\texttt{FILTER}\; (P(\bar{x}) \vee Q(\bar{y}))$ as syntactic sugar for $(\varphi \;\texttt{FILTER}\; P(\bar{x})) \;\texttt{FILTER}\; Q(\bar{y})$ or $(\varphi \;\texttt{FILTER}\; P(\bar{x})) \;\texttt{OR}\; (\varphi \;\texttt{FILTER}\; Q(\bar{y}))$, 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 $\mathrm{var}(\varphi)$ the set of all variables mentioned in φ (including filters), and by $\mathrm{vdef}(\varphi)$ all variables defined in φ by a clause of the form $R \;\texttt{AS}\; x$. Furthermore, $\mathrm{vdef}_+(\varphi)$ denotes all variables in $\mathrm{vdef}(\varphi)$ 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 $\mathrm{var}(\varphi) = \{x, y, z\}$, $\mathrm{vdef}(\varphi) = \{x, y\}$, and $\mathrm{vdef}_+(\varphi) = \{x\}$. 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:

- $\varphi = R \;\texttt{AS}\; x$, $C = \{\nu(x)\}$, and $\mathrm{type}(S[\nu(x)]) = R$.
- $\varphi = \psi \;\texttt{FILTER}\; P(x_1, \ldots, x_n)$, $C \in [\![\psi]\!](S, \nu)$ and $(S[\nu(x_1)], \ldots, S[\nu(x_n)]) \in P$.
- $\varphi = \psi_1 \;\texttt{OR}\; \psi_2$ 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 $U = \mathrm{vdef}_+(\psi)$.

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.

NEXT. 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 ($C_1 \triangle C_2$) 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 $\min(C_1 \triangle C_2) \in C_2$. For example, $\{5,8\} \leq_{\text{next}} \{1,8\}$ since the minimum element in $\{5,8\} \triangle \{1,8\} = \{1,5\}$ is 1, which is in $\{1,8\}$. Note that 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 $C \in [\![\texttt{NXT}(\varphi)]\!](S,\nu)$ 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:

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

where SEL is any selection strategy. We say that a CEL formula φ is *well-formed* if for every subformula of the form $\psi \texttt{ FILTER } P(\bar{x})$ 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 $(T \texttt{ AS } y \texttt{ FILTER } y.id = x.id)+$ 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 $\psi = T \texttt{ AS } x \,;\, T \texttt{ AS } x$ 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:

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

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 denote by $\mathrm{Run}_n(\mathcal{A}, S)$ the set of accepting runs of $\mathcal{A}$ over S of length n. Further, $\mathrm{events}(\rho)$ is the set of positions where the run *marks* S, namely $\mathrm{events}(\rho) = \{i \in [0, n] \mid m_i = \bullet\}$. 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 $[\![\mathcal{A}]\!]_n(S) = \{\mathrm{events}(\rho) \mid \rho \in \mathrm{Run}_n(\mathcal{A}, S)\}$ 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 labelled by `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,

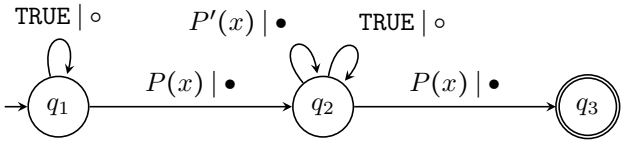

Figure 2 A CEA that can generate an unbounded amount of complex events. Here $P(x) := \text{type}(x) = H$ and $P'(x) := \text{type}(x) = T \wedge x.temp > 40$.

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' `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 6.** *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].

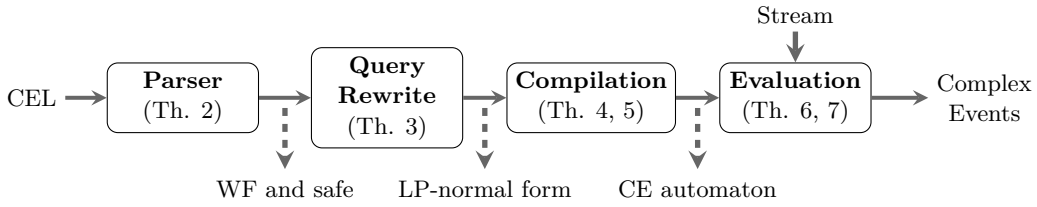

Figure 3 Evaluation framework for CEL.

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 6), 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 7.** *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 8.** *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* $\texttt{SEL} = \texttt{MAX}$.

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 7). 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 8 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. Accessed on 2018-01-05. URL: `http://www.espertech.com/`.

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

4 Asaf Adi and Opher Etzion. Amit-the situation manager. *VLDB Journal*, 2004.

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

6 Alfred V. Aho. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

7 Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *VLDB*, 2008.

8 Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In *RR*, 2010.

9 Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *SIGMOD*, 2003.

10 Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 2006.

11 Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex Event Recognition Languages: Tutorial. In *DEBS*, pages 7–10. ACM, 2017.

12 Alexander Artikis, Marek Sergot, and Georgios Paliouras. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):895–908, 2015.

13 Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. Logic-based event recognition. *The Knowledge Engineering Review*, 27(4):469–506, 2012.

14 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*, 2007.

15 Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*, 2007.

16 Jean Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

17 Alejandro Buchmann and Boris Koldehofe. Complex event processing. *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 2009.

18 Jan Carlson and Björn Lisper. A resource-efficient event algebra. *Science of Computer Programming*, 2010.

19 Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

20 Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. A logic-based, reactive calculus of events. *Fundamenta Informaticae*, 105(1-2):135–161, 2010.

21 Gianpaolo Cugola and Alessandro Margara. Raced: an adaptive middleware for complex event detection. In *Middleware*, 2009.

22 Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In *DEBS*, 2010.

23 Gianpaolo Cugola and Alessandro Margara. Complex Event Processing with T-REX. *The Journal of Systems and Software*, 2012.

24 Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 2012.

25 Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. A general algebra and implementation for monitoring event streams. Technical report, Cornell University, 2005.

26 Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

27 Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In *DEXA*, 2002.

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

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

30 Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *ICDE 2008*, pages 1391–1393. IEEE, 2008.

31 Yeye He, Siddharth Barman, and Jeffrey F. Naughton. On Load Shedding in Complex Event Processing. In *ICDT*, pages 213–224, 2014.

32 Yeye He, Siddharth Barman, Di Wang, and Jeffrey F Naughton. On the complexity of privacy-preserving complex event processing. In *PODS*, pages 165–174, 2011.

33 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

34 Elena Ikonomovska and Mariano Zelke. Algorithmic Techniques for Processing Data Streams. *Dagstuhl Follow-Ups*, 2013.

35 Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay 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 Masoud Mansouri-Samani and Morris Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 1997.

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

39 Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. Network intrusion detection. *IEEE network*, 1994.

40 Peter Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Middleware*, 2003.

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

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

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

44 Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

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

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

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

48 Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. What is next in event processing? In *PODS*, pages 263–272, 2007.

49 Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

50 Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, 2014.

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