ABSTRACT

Complex Event Processing (CEP) has emerged as the unifying field for technologies that require processing and correlating heterogeneous 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 frameworks are based on ad-hoc solutions that do not rely on solid theoretical ground, making them hard to understand, extend, or generalize. Moreover, they are usually presented as application programming interfaces documented by examples, and using each of them requires learning a different set of skills.

In this paper we embark on the task of giving a rigorous framework to CEP. As a starting point, we show the current practical drawbacks of having CEP systems that do not rely on solid foundations. Then, we propose a formal language for specifying complex events, called CEPL, that contains the common features used in the literature and has a simple and denotational semantics. We also formalize the so-called selection strategies, which are the cornerstone of CEP and had only been presented as by-design extensions to existing frameworks. With a well-defined semantics at hand, we study how to efficiently evaluate CEPL for processing complex events. We provide optimization results based on rewriting formulas to a normal form that simplifies the evaluation of filters. Furthermore, we introduce a formal computational model for CEP based on transducers and symbolic automata, called match automata, that captures the regular core of CEPL, i.e., formulas with unary predicates. By using rewriting techniques and automata-based translations, we show that formulas in the regular core of CEPL can be evaluated using constant time per event followed by constant-delay enumeration of the output (under data complexity). By gathering these results together, we propose a framework for efficiently evaluating CEPL, establishing solid foundations for future CEP systems.

1. INTRODUCTION

The problem of automatically processing continuously arriving information has been present in the database community since the conception of the first Database Management Systems. The so-called Active Database Systems (ADBMS) [35] presented a first attempt to solve this problem by allowing users to write triggers that are executed upon arrival of tuples. The main goal of ADBMSs was to provide integrity and persistence, focusing on secondary storage (see, e.g., [46, 20]). Naturally, this made ADBMSs poor in terms of performance. Data Stream Management Systems (DSMS) were introduced to work on main memory and overcome this limitation [30]. Like traditional database management systems, DSMSs are concerned with executing relational queries but over dynamic data (see for example [21, 6, 14]), and maintaining a live version of the results over time. Since DSMSs focus on relational queries over streams, they offer limited reactive capabilities and only see streams as data arriving by parts, and not as a sequence of events [25].

Modern applications must rapidly react to data arriving in high-throughput environments. Moreover, in scenarios like Network Intrusion Detection [37], Industrial Control Systems [31] or Real-Time Analytics [41], streams must be seen as ordered data events, giving high importance to the order in which the information arrives. Since ADBMSs and DSMSs only fulfill these requirements partially, different communities have proposed domain-specific frameworks and tools for dealing with their particular needs.

Complex Event Processing (CEP) has emerged as the unifying field of technologies for the aforementioned scenarios. From a general perspective, the main requirement of a CEP framework is detecting situations of interest under high-throughput streams. Prominent examples of CEP systems include Sase [46], Cayuga [26], Amit [8] and CEDR [17], among others (see [25] for a good survey). With the objective of making CEP systems applicable to real-life situations, issues like scalability, fault tolerance and distribution have been the main focus of these systems. Other design decisions, like query languages, are generally adapted to match computational models that can efficiently process data (see for example [45]). This has produced new data management and optimization techniques, generating promising results in the area [20, 17, 24].

Unfortunately, existing CEP systems are based on ad-hoc solutions that do not rely on solid theoretical ground, usually presenting solutions for particular domains. It is hard to find a common theoretical ground, which makes CEP frameworks difficult to understand, extend, or generalize. More specifically, the main problems of current CEP frameworks are the following.

First, as has been claimed several times [28, 49, 23]...
the languages for detecting complex events over streams generally lack well-defined denotational semantics. The semantics of several languages are defined either by examples [34, 8, 22], or by intermediate automata models [46, 43, 39]. Although there are frameworks that introduce formal semantics (e.g., [26, 17, 11, 23, 13]), they do not meet the expectations to pave the foundations of CEP languages. For instance, some of them are too complicated (e.g., sequencing is combined with filters), have unintuitive behavior (e.g., sequencing operator is non-associative), or are severely restricted (e.g., only basic operations are supported). As an example, iteration is a fundamental operator in CEP and 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 [40, 29]. Thus, without a formal and natural semantics the languages for CEP are in general cumbersome.

The lack of simple denotational semantics also makes it hard to introduce general query optimization techniques. It is common to find complicated heuristics and optimizations that cannot be replicated in other frameworks (see e.g., [48]). Furthermore, optimizations are usually proposed at the architecture level [33, 26, 39], preventing a unifying optimization theory. This also makes it hard to leverage ideas like query rewriting, a well-developed technique in database management systems [7, 10]. An exception here is [43], which uses limited query rewriting techniques.

Another limitation of existing CEP frameworks is that, for query evaluation, they used ad-hoc automata models [26, 17, 11] without considering previous advances in automata theory [42]. These models are usually complicated [39, 43], informally defined [26], or non-standard [23, 9]. For instance, some CEP frameworks enhanced non-deterministic finite state automata with predicates [43, 39], buffers [9], functions [13], time intervals [39], etc. Most of these features have been studied before in automata theory [42, 45, 12], yet they are defined in CEP frameworks without considering previous results in the field. In practice this implies that, although finite state automata is a recurring approach in CEP, there is no common model in the CEP literature.

Given this scenario, the main goal of this paper is to give solid foundations to CEP systems in terms of the query language and query evaluation. Towards these goals, our contributions can be divided in three parts. In the first part, we show that the lack of solid theoretical foundations make current CEP systems unable to handle even basic workloads. Indeed, we experimentally show that current CEP systems cannot evaluate basic queries efficiently. The second part is dedicated to provide a formal language that allows for expressing the most common features of CEP systems, namely sequencing, filtering, disjunction, and iteration. To this end we introduce CEP-logic (CEPL for short), a logic with well-defined compositional and denotational semantics. We also formalize the notion of selection strategies which is usually discussed directly [48] or indirectly [17] in the literature but has not been formalized at the language level.

Finally, in the third part we embark on the design of a formal framework for CEPL evaluation. This framework must consider three main building blocks for the efficient evaluation of CEPL: (1) syntactic techniques for rewriting CEPL queries, (2) a well-defined intermediate evaluation model, and (3) efficient translation and algorithms to evaluate this model. Regarding the rewriting techniques, we study the structure of CEPL by introducing natural syntactic restrictions (well-formed and safe formulas), and show that these restrictions are relevant for the complexity of query evaluation. Further, we give a general result on rewriting CEPL 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 CEPL, called match automata, which is based on transducers and symbolic automata. We show that this model has good properties (e.g. it is closed under determinization) and study the evaluation of match automata by showing efficient algorithms for running any match automata with constant time per tuple followed by constant-delay enumeration of the output. We also provide translations for any CEPL formula into match automata.

Finally, we bring together our results to present a formal framework for evaluating CEPL, and pinpoint where the main issues for efficient evaluation reside.

Organisation. We give an intuitive introduction to CEP and our framework in Section 2. In Section 8 we show evidence of the practical drawbacks of having CEP systems that do not rely on solid theoretical foundations. Our logic and the selection strategies are formalized in Section 4 and 5 respectively. The syntactical structure of the logic is studied in Section 6 and the computational model and its properties are given in Section 7. Section 8 is devoted to show how the logic can be evaluated through match automata. Section 9 puts all the results in perspective and presents an efficient evaluation framework for CEP Systems. We finally give some concluding remarks in Section 10. Due to space limitations all proofs are deferred to the appendix.

2. EVENTS IN ACTION

In this section we motivate and present the main features and challenges of CEP. The examples used in this section will also serve throughout the paper as running examples.

In a usual CEP setting, events arrive in a streaming fashion to a system that must detect certain patterns [25]. For the purpose of illustration assume there is a stream produced by wireless sensors positioned in a farm, whose objective is both to detect fires and achieve optimal irrigation. For the sake of simplification, assume that there are three sensors, and each of them can measure both temperature (in Celsius degrees) and relative humidity (as 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...
Figure 1: A stream \( S \) of events measuring temperature \((T)\) and humidity \((H)\). “value” contains degrees and humidity for \(T\)- and \(H\)-events, respectively.

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. We present such a stream in Figure 1 where each column represents an event and the value row is the temperature or relative humidity if the event is of type temperature \((T)\) or humidity \((H)\), respectively.

As previously mentioned, complex events are generally specified by domain experts in the form of patterns. 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. Then, such sequence of two events is a complex event of interest. Let us intuitively explain the syntax and semantics with which a domain expert can express this as a pattern (from now on a formula) in our framework:

\[
\varphi_1 = (T \text{ as } x; H \text{ as } y) \text{ FILTER } (x.tmp > 40 \land y.hum <= 25 \land x.id = 0 \land y.id = 0)
\]

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 and low humidity measured by sensor 0. Before defining the semantics of \( \varphi_1 \), let us discuss what would be the expected result of evaluating this formula over a stream. A first important remark is that event streams are noisy in practice, 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 (as opposed to regular expressions). 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 events matching a formula are particularly relevant to the end user. Therefore, every time that a formula matches a complex event in the stream, the final user should obtain enough information to retrieve the events that compose the complex event. Therefore, the output of evaluating a formula over a stream is a set of matches, where each match is the set of indexes (stream positions) of the events that witness the complex event.

We proceed to intuitively explain the evaluation of \( \varphi_1 \) over the stream \( S \) (Figure 1). Let \( S[i] \) be the event with index \( i \) in the stream. What we expect as output is a set of pairs \( \{i, j\} \) such that \( S[i] \) is of type \( T \), \( S[j] \) is of type \( H \), \( i < j \), and they satisfy the conditions expressed in the FILTER. By inspecting this stream, we can see that the pairs satisfying these conditions are \( \{1, 2\} \), \( \{1, 8\} \), and \( \{5, 8\} \). These are the elements that the user should get as output in order to retrieve the events from the stream for further analysis.

Formula \( \varphi_2 \) illustrates in a simple way the two most elemental features of CEP, namely sequencing and filtering \([25]\). But although it detects a set of possible fires, it restricts the order in which the two events must occur, namely 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 \([25]\). In our framework, disjunction is expressed by means of the OR operator. To illustrate, we extend \( \varphi_1 \) by allowing events to appear in arbitrary order.

\[
\varphi_2 = [(T \text{ as } x; H \text{ as } y) \text{ OR } (H \text{ as } y; T \text{ as } x)] \text{ FILTER } (x.tmp > 40 \land y.hum <= 25 \land x.id = 0 \land y.id = 0)
\]

Intuitively, the OR operator allows for any of the two patterns to be matched, and then applies the filter as in \( \varphi_1 \). The result of evaluating \( \varphi_2 \) over the stream \( S \) of Figure 1 is the same as evaluating \( \varphi_1 \) over \( S \) plus the match \( \{2, 5\} \).

So far we have illustrated the use of CEP as a mean to raise alerts when a certain complex event occurs, but 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 whenever there is a sudden increase of humidity. A problem here is that we don’t know a priori the amount of temperature measurements, and therefore we need to capture an unbounded amount of events. An operator for iteration \([25]\), commonly denoted by +, is generally introduced in CEP frameworks for solving this problem. The + operator introduces several difficulties in the semantics of CEP languages. For example, since events are not required to occur continguously in a stream, the nesting of + is particularly tricky and most frameworks simply disallow this (see for example \([40]\)). Coming back to our example, the formula for measuring temperatures whenever a sudden increase of humidity is detected by sensor 1 is:

\[
\varphi_3 = [H \text{ as } x; (T \text{ as } y \text{ FILTER } y.id = 1) +; H \text{ as } z] \text{ FILTER } (x.hum <= 30 \land z.hum > 60 \land x.id = z.id = 1)
\]

Intuitively, variables \( x \) and \( z \) witness a sudden 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 (see, e.g., \([40]\)). Although it is possible to define the semantics for that syntax (simply as a universal quantifier over the occurrences of the variable), 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 \((T \text{ AS } y \text{ FILTER } y.id = 1)\) an unbounded number of times: do we want to associate y to one event or to different events across repetitions? Certainly, we want the latter option since each of the matched temperatures \((i.e. T \text{ events})\) will be different. In Section 4 we introduce a natural semantics that allows for nesting arbitrarily many + and associate variables (inside + operators) to different events across repetitions.

Now let us explain the semantics of \( \varphi_3 \) over stream \( S \) (Figure 1). First, notice that the only two humidity events satisfying the top-most filter are \( S[3] \) and \( S[7] \). The temperature measurements between these two events are \( S[4] \) and \( S[6] \). As expected, the match \( \{3, 4, 6, 7\} \) is part of the output. However, there are also other matches in the output. Since, as discussed, there might be irrelevant events between relevant ones, the semantics of + must allow for skipping arbitrary events. Actually, in the presented match we are already skipping some humidity and temperature events. This implies that, in order to provide well-defined compositional semantics, one must allow for skipping events that might be of interest (in our example, temperature measurements of sensor 1). Therefore, the matches \( \{3, 6, 7\} \) and \( \{3, 4, 7\} \) are also part of the output.

The set of matches generated by formulas \( \varphi_1 \) and \( \varphi_3 \) raises an interesting question: are users interested in receiving all matches? Are some matches more informative than others? Coming back to the output matches of \( \varphi_3 \) \( \{3, 6, 7\}, \{3, 4, 7\} \) and \( \{3, 4, 6, 7\} \), one can easily argue that the biggest match is more informative than others since all matches are contained in it. A more complicated analysis deserves the matches output by \( \varphi_1 \). 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. Given that \( \{1, 8\} \) happens before \( \{5, 8\} \), a user would probably want \( \{1, 8\} \) as the only output match of \( \varphi_1 \) when the last event of the stream is received. The decision of generating only a subset of the matches, and which subset to return, is generally called a selection strategy [46, 48]. A common design across CEP-systems is that formulas are defined to extract all matches and it is responsibility of the users to apply selection strategies over formulas to restrict the set of output matches. Selection strategies are a fundamental feature of CEP but, unfortunately, there is no previous proposal that has defined them formally. A special mention deserves the so-called next selection strategy [46, 48] which in CEP systems usually models the idea of outputting the “most consecutive” match. Although the semantics of next has been proposed or mentioned in previous papers \( (e.g \ [17]) \), it is usually defined incorrectly [46, 48] or across the language making simple operators complicated [26]. In Section 5 we formally define selection strategies, including next. Furthermore, we show in Section 6 that the next selection even allows to optimize the evaluation of formulas.

As it can be deduced from the examples above, the evaluation of CEP formulas can easily become computationally intensive. For example, the + operator allows for a power-set construction, potentially introducing an exponential blowup in the number of results. Therefore, the effectiveness of a CEP framework is based not only on the expressive power of its formulas, but also on the efficiency with which formulas can be evaluated. Because of their similarities with regular expressions, it is common to find automata-based models for evaluating CEP formulas in the literature [26, 17, 11]. In Section 7 we introduce a model named match automata that is based on synchronized transducers [24] and symbolic automata [45]. We also provide a translation from CEP formulas like the ones presented above to match automata.

When evaluating a match automaton over a stream, an important optimization is to stop the runs that will not lead to a match as soon as possible. To illustrate this fact, consider again formula \( \varphi_1 \). 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 \( id = 0 \) and whose temperature measurement is more than 40 degrees. Only afterwards the possible matching events \( y \) should be considered. In Section 6 we present rewriting techniques for CEP formulas that allow for this type of optimization. In particular, we present a procedure for pushing filter conditions as close to the definition of the variables as possible. For example, formula \( \varphi_1 \) can be restated as

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

In this case the translation is straightforward because the FILTER condition of \( \varphi_1 \) only contains conjunctions. However, when adding logical disjunction the rewriting needs a more involved analysis of the formula.

We conclude this section by illustrating 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 a sudden increase of humidity there. Then, what we need is a pattern similar to \( \varphi_3 \) where all the events must be produced by the same sensor, but that sensor is not necessarily sensor 1. This is achieved by the following pattern:

\[
\varphi_4 = [H \text{ AS } x; (T \text{ AS } y \text{ FILTER } y.id = x.id) +; H \text{ AS } z]
\]

FILTER \( (x.hum < 30 \land z.hum > 60 \land x.id = z.id) \)

Notice that here the filters contain the binary 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, formula \( \varphi_4 \) is more complicated in the sense that the value of \( x \) must be remembered and used during the evaluation in order to compare it
with all the incoming events. If the reader is familiar with automata theory, this behavior is clearly not “regular” and it will not be captured by a finite state model. In this paper, we want to study and characterize the regular part of CEP-systems. Therefore, in Sections 7, 8 and 9 we restrict our analysis to formulas with unary predicates (like $\varphi_1$, $\varphi_2$, and $\varphi_3$) that capture the regular core of CEP-languages, and postpone the analysis of formulas like $\varphi_4$ for future work. It is important to mention here that the semantics of our language proposal (plus the selection strategies and rewriting of formulas) is defined in general and not restricted to any subfragment.

We have illustrated sequencing, filtering, disjunction, iteration and correlation, and we discussed optimization techniques and features of CEP that will be further developed in the rest of the paper. In the next section we proceed to formally define the syntax and semantics of CEP formulas.

### 3. DO WE NEED A NEW FRAMEWORK?

Before going into the technical details of our proposal, a natural question to ask at this point is whether we need a new formal framework for complex event processing. In this direction, one could easily argue that the area of CEP is mature enough: in the last twenty years, many frameworks and commercial products have been presented (see [3] for a good overview). Furthermore, we intend to formalize the most basic operators and functionalities of CEP systems and, given the long list of existing systems, this might seem like working over a “folklore result” in the area of CEP.

To support our position with respect to the need of a new formal CEP framework, in this section we inspect the query languages and the performance of well-established academic and commercial CEP systems. These frameworks are TESLA/T-Rex [23, 24] (version of February 2017 [5]), SASE [17] (version of August 2014 [3]), Esper Enterprise Edition [1] (version 6.0.1) and Oracle Stream Analytics [2] (version 12.2.1.1). T-Rex and SASE are two of the most cited systems in CEP [25], and Esper and OracleSA are leading commercial products in CEP market [3]. Note that T-Rex, SASE, and Esper are CEP systems but OracleSA is, strictly speaking, more a Data Stream Management System (DSMS) [25]. Despite of this difference, we included OracleSA in our tests because it implements CQL [15]. This is one of the most cited languages in DSMS, making our thesis stronger with respect to query languages for data stream processing.

We show that all of these systems lack from a full implementation of the basic CEP operations and, moreover, they perform poorly even when evaluating a simple sequencing query. The most common features proposed in the literature [25] are filtering (FILTER), disjunction (OR), sequencing (;) and iteration (+). The following table is a summary of the operations supported by each of the considered systems:

| System   | Filtering | OR | Sequencing | Iteration |
|----------|-----------|----|------------|-----------|
| T-Rex    | Limited   | No | Limited    | No        |
| SASE     | Limited   | No | Yes        | Yes       |
| Esper    | Yes       | Yes| Yes        | Limited   |
| OracleSA | Yes       | Yes| No         | Limited   |

Although these are well-established systems in the area, none of them supports all the basic operations of CEP. Moreover, there are operators (e.g. iteration) that are not fully supported by any system, showing a lack of understanding of its formal semantics. This suggests that, without a good definition of the syntax and semantics of the query language, it is very difficult to implement all operators correctly, suggesting that a formal framework for CEP is needed.

Next, we show that current CEP systems are not only lacking support for certain operators, but also that they perform poorly even under simple workloads. Given that there is no operator that is fully supported by every system, we concentrate on testing the performance with an essential operation of CEP, namely sequencing. It is widely accepted that this is the most fundamental operator in CEP, and therefore any system should evaluate it efficiently. For our experiments we consider the next query:

$$\varphi_c = (H \text{ AS } T \leq e; H \text{ AS } y; H \text{ AS } z; T \text{ AS } t).$$

Following our running example, this query simply asks for three humidity events followed by a temperature event. It is important to mention that, although OracleSA does not support sequencing (see the table above), for $\varphi_c$ we can assign a number to each tuple in increasing order and use $\leq$-comparison to emulate sequencing. T-Rex directly supports this simple form of sequencing.

To test the performance over streams of different sizes, we evaluate $\varphi_c$ over streams of the form $S_n = H^e T$ where $n$ events of type $H$ are followed by a single event of type $T$. The output of $\varphi_c$ over $S_n$ are all possible combinations of three $H$-events plus the last $T$-event. Given that the output of this will be cubic in $n$, in our experiments we measure the running time starting from the first event in $S_n$ until the first output is received. The testing was done using a Laptop with an Intel Core i5-6200u processor and 8 GB of main memory running Windows 10. Each experiment was repeated three times and the average score was reported (although there were no significant deviations from the average).

Figure 2 depicts the obtained results. Missing measurements indicate that the corresponding system took more than two minutes before generating the first output. It is worth noticing that, with the exception of T-REX, in the case of missing values the systems had not even read the final $T$ event of the stream after the two minutes. Regarding the actual results, a first important remark is that no system was able to process streams of 500 tuples. Moreover, they take more than 1 second

---

1 Only filters connected with conjunctions are supported.
2 A window-based sequencing operator is supported.
3 Supported over atomic events and no nesting is allowed.
4 Every iteration requires a termination condition.
Given a stream \( S = t_0t_1 \ldots \) 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 \( S_i \). Here, we suppose that the order of the sequence implicitly defines an order among tuples and we usually call \( S[i] \) an event of \( S \) at time \( i \). Furthermore, contrary to other frameworks \[9\] we consider that the time of each event is implicitly given by the order of the stream and we do not consider extensions like intervals. We leave these extensions for future work (see Section 10).

Let \( X \) be a set of variables and \( \mathbf{P}(R) \) a set of predicates over tuples (\( R \)), where each \( P \in \mathbf{P}(R) \) has arity (\( P \)). For the sake of simplification, for each \( P \in \mathbf{P}(R) \) we write \( P(x_1, \ldots, x_n) \), where \( n = \text{arity}(P) \) and \( x_1, \ldots, x_n \in X \). We define the set \( \mathbf{F}(R) \) of selection formulas over \( R \) as the smallest set of formulas such that \( \mathbf{P}(R) \subseteq \mathbf{F}(R) \) and is closed under conjunction, disjunction and negation. For example, if \( \mathbf{P}(R) \) contains the predicates \( P_1(z) := z.\text{hum} < 30 \), \( P_2(z) := z.\text{hum} > 60 \) and \( P_3(x, z) := z.\text{id} = z.\text{id} \), then the outer-most filter of \( \varphi_4 \) (see Section 2) is a formula in \( \mathbf{F}(R) \).

An assignment is a partial function \( \sigma : X \rightarrow \text{tuples}(R) \). Given an assignment \( \sigma \) and a predicate \( P(x_1, \ldots, x_n) \) in \( \mathbf{P}(R) \), we say that \( \sigma \) satisfies \( P \) (denoted by \( \sigma \models P \)) if \( P(\sigma(x_1), \ldots, \sigma(x_n)) \) evaluates to true. For every formula in \( \mathbf{F}(R) \) that is not a predicate, the semantics is defined recursively as usual. Finally, for the computational complexity analysis we assume that given an assignment \( \sigma \) and a \( \alpha \in \mathbf{F}(R) \), it takes time \( O(1) \) to verify whether \( \sigma \models \alpha \).

### 4.2 Core CEP Logic

Now we proceed to give the syntax of what we call the core of CEPL (core-CEPL for short), a logic inspired by previous CEP frameworks \[16, 20, 17\]. This language features those operations commonly found in the literature.

The set of formulas in core-CEPL, or core formulas for short, is given by the following grammar:

\[
\begin{align*}
\varphi &::= R \text{ AS } x \mid \varphi \text{ FILTER } \alpha \mid \varphi \text{ OR } \varphi \mid \varphi;\varphi \mid \varphi^+
\end{align*}
\]

where \( R \) is a relation name, \( x \) is a variable in \( X \) and \( \alpha \) is a selection formula in \( \mathbf{F}(R) \). As opposed to existing frameworks, we do not restrict the use of variables, or nesting of operators. In particular, we allow for arbitrary nesting of +.

Now we proceed to define the semantics of core formulas, for which we need to introduce some further notation. A match \( M \) is defined as a non-empty and finite set of natural numbers. As mentioned in the previous section, a match contains the positions that witness the satisfaction of a formula over a stream, and moreover, they are the final output of evaluating a formula over a stream. We denote by \( |M| \) the size of a match \( M \) and by \( \min(M) \) and \( \max(M) \) the minimum and maximum element of \( M \), respectively. Given a stream \( S = t_0t_1 \ldots \) and \( M = \{i_1, i_2, \ldots, i_n\} \) with \( i_1 < i_2 < \ldots < i_n \), the subsequence \( t_{i_1} t_{i_2} \ldots t_{i_n} \) of \( S \) is denoted by \( S[M] \). Intuitively, if \( S[i] \) is an event of \( S \), \( S[M] \) represents a

![Figure 2: Processing time before producing the first output to \( \varphi \) over H"our \( v/s n \) (stream size).](image-url)
There are a couple of important remarks here. First, notice that the valuation $\nu$ can be defined over a superset of the variables mentioned in the formula. This is important for the sequencing operator (\'\) because we require the matches from both sides to be produced with the same valuation. Second, when we evaluate a subformula of the form $\rho+$, we carry the value of variables defined outside the subformula. For example, the subformula $(T \ AS \ y \ FILTER \ y.id = x.id)+$ of $\varphi_4$ 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 in the upper level). 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 in $\varphi_4$.

Notice also that the sequencing operator (\') is associative. Although this might seem natural, there are CEP frameworks with formal semantics in which this is not the case (see, e.g., [26]). This is one of the reasons to include the position $i$ in our definition, as it restricts the matches produced by the right-hand side of a sequence only to those occurring after the left-hand side was matched. Also, this will allow us to give compositional semantics to selectors (Section 5).

As it was previously mentioned, in a core-CEPL formula variables are just used for comparing attributes with FILTER and are not relevant for the final output. To this end, we say that $M$ belongs to the evaluation of $\varphi$ over $S$, denoted by $M \in [\varphi](S)$, if there exists a valuation $\nu$ such that $M \in [\varphi](S,0,\nu)$, namely, we evaluate $\varphi$ over $S$ starting from position 0. As an example, the reader can check that the matches presented in Section 2 are indeed matches of $\varphi_1$ to $\varphi_3$ over the stream of sensors measurements.

### 4.3 Other operators

We now extend the syntax and semantics of core-CEPL by adding new operators. Some of these operators are natural extensions of the core language and others have been proposed in previous work [17, 8]. Specifically, the syntax of the extended core CEPL (or ecore-CEPL) is given by extending the grammar of core-CEPL with the following operators:

$$
\varphi := \varphi \ AND \ \varphi \ | \ \varphi \ ALL \ \varphi \ | \ \varphi \ UNLESS \ \varphi
$$

In this case we call $\varphi$ an ecore-CEPL formula (or simply ecore formula).

Similar to core-CEPL, we define the semantics of ecore-CEPL over a stream $S$ by using the notions of matches and valuations. The semantics of the core operators are as defined in Section 4.2 and the semantics of a formula $\varphi = \rho_1 \ OP \ \rho_2$, where $OP$ is any of the new operators AND, ALL, and UNLESS, is defined recursively as follows: given a match $M$, a stream $S$, a position $i$ and a valuation $\nu$, we say that $M \in [\varphi](S,i,\nu)$ if one of the following conditions holds:

- $\varphi = R \ AS \ x$, $M = \{\nu(x)\}$, $type(S[\nu(x)]) = R$ and $i \leq \nu(x)$.
- $\varphi = \rho \ FILTER \ \alpha$, $M = [\rho](S,i,\nu)$ and $\nu_S = \nu$.
- $\varphi = \rho_1 \ OR \ \rho_2$ and $(M \in [\rho_1](S,i,\nu)$ or $M \in [\rho_2](S,i,\nu))$.
- $\varphi = \rho_1$ and $M = M_1 \cdot M_2$ for two matches $M_1$ and $M_2$ such that $M_1 \in [\rho_1](S,i,\nu)$ and $M_2 \in [\rho_2](S,j,\nu)$ with $j = max(M_1) + 1$.
- $\varphi = \rho+$ and there is a valuation $\nu'$ such that either $M \in [\rho](S,i,\nu' \rightarrow U)$ or $M \in [\rho; \rho+](S,i,\nu' \rightarrow U)$, where $U = \nu_{def}(\rho)$.

There are a couple of important remarks here. First,
Recall that formula $\varphi$ is a valuation $\nu$ to process large numbers of events. This can be expressed by the following pattern:

$$[(T \text{ AS } x) \text{ FILTER } (x \text{. tmp} < 20)];
(T \text{ AS } y) \text{ FILTER } (y \text{. tmp} > 40)] \text{ UNLESS}
[(T \text{ AS } z) \text{ FILTER } (z \text{. tmp} >= 20 \land z \text{. tmp} <= 40)]$$

We stress that valuations are not part of the output, contrary, ALL is more flexible and allows to combine two matches. In this sense, ALL is similar to sequencing but allows that the matches occur at any point in time, even overlapping and intersecting. For example, formula $\varphi_2$ of Section 2 asks for a temperature measurement and a humidity measurement that can occur in any order and satisfy a certain condition. This formula could have been written more succinctly as $[(T \text{ AS } x) \text{ ALL } (H \text{ AS } y)] \text{ FILTER } (\ldots)$. The objective of the UNLESS is to introduce negation. It is important to mention that the negated formula (the right-hand side) is restricted to matches between the start and end of matches for the formula in the left-hand side. This is motivated by the fact that a match should not depend on objects that are distant in the stream. For example, consider that we want to see a drastic increase in temperature. This can be expressed as a sequence of a low temperature (less than 20 degrees) and a high temperature (more than 40 degrees), where no other temperatures occur in between. This can be expressed by the following pattern:

$$(T \text{ AS } x) \text{ FILTER } (x \text{. tmp} < 20) ;
(T \text{ AS } y) \text{ FILTER } (y \text{. tmp} > 40)] \text{ UNLESS}
[(T \text{ AS } z) \text{ FILTER } (z \text{. tmp} >= 20 \land z \text{. tmp} <= 40)]$$

5. SELECTION STRATEGIES

Matching complex events is usually a computationally intensive task. As our running example and the definition of ecore-CEPL might suggest, the main reason behind this is that the amount of matches can grow exponentially in the size of the stream, forcing systems to process large numbers of candidate matches. In order to optimize the matching processes, it is common to restrict the set of results [25, 46, 48]. This is one of the cornerstones of CEP systems, and most of the proposals in the literature introduce them simply as ad-hoc extensions matching particular computational models. For a more general approach, we introduce the so-called selection strategies as unary operators (called selectors).

Formally, we define the syntax of full CEPL, or simply CEPL, by the following grammar:

$$\varphi ::= R \text{ AS } x \mid \varphi \text{ FILTER } \alpha \mid \varphi \text{ OR } \varphi \mid \varphi; \varphi \mid \varphi + \varphi \text{ AND } \varphi \mid \varphi \text{ ALL } \varphi \mid \varphi \text{ UNLESS } \varphi \mid \text{STRICT}(\varphi) \mid \text{NXT}(\varphi) \mid \text{MAX}(\varphi)$$

We now proceed to define the semantics of the selectors, starting by the strict-contiguity selector $\text{STRICT}$. Recall that formula $\varphi_1$ in Section 2 detects complex events composed by a temperature above 40 degrees Celsius followed by a humidity of less than 25%. As already argued, in general one could expect many other events between $x$ and $y$. However, it could be the case that this particular pattern is of interest only if the events occur contiguously in the stream, namely a temperature right after a humidity measure. The strict-contiguity selector $\text{STRICT}$ only allows strictly consecutive matches. Formally, for any CEPL formula $\varphi$ we have that $M \in [\text{STRICT}(\varphi)](S, i, \nu)$ holds if $M \in [\varphi](S, i, \nu)$ and for every $i, j \in M$ such that $i < k < j$ (i.e., $M$ is an interval). For example, in our running example $\text{STRICT}(\varphi_1)$ would only produce the match $\{1, 2\}$, although $\{1, 8\}$ and $\{5, 8\}$ are also matches for $\varphi_1$ over $S$. The $\text{STRICT}$ contiguity selector is generally included in CEP frameworks because it allows to carry over good properties from regular expressions. However, for reasons we have already discussed it is not particularly interesting to capture contiguous events in streams. In CEP one would like to have more flexible selection strategies that allow for obtaining fewer yet meaningful results.

The other notion that appears often in the literature is that of selecting only those matches that are as consecutive as possible. For example, consider again the pattern $\varphi_1$ and the stream $S$ from Section 2. As we discussed, both $\{1, 8\}$ and $\{5, 8\}$ are matches for $\varphi_1$ over $S$. Now, if a user didn’t want to obtain all matches, which of these two matches would he prefer? Here is where the notion of “most consecutive” appears. It is widely accepted that the first match is preferred over the second, since the first event of $\{1, 8\}$ occurred before the first event of $\{5, 8\}$, and therefore the first match is more consecutive in the stream. Another intuition behind this notion is that it is better to match the next event instead of skipping it and selecting another event in the future. The above example motivates the semantics of the next selection strategy, which has been discussed and used in most of the CEP systems [25] as an special operator [17, 46, 48] or behind the semantics of the sequencing operator [26]. However, they fail to define it correctly since they either mix the semantics of selectors with the semantics of sequencing or iteration, or they define the selector semantics for a restricted set of operators (e.g., they cannot support Kleene closure). In our framework, we formalize the semantics of the next operator based on a special order over matches. This allows us to capture the intuition of “more consecutive” reflected in the literature, while giving a general definition.

Let $M_1$ and $M_2$ be two matches. The symmetric difference between $M_1$ and $M_2$ is denoted by $M_1 \triangle M_2$ and is the set of all elements either in $M_1$ or $M_2$ but not in both. We say that $M_1 \preceq_{\triangle} M_2$ if either $M_1 = M_2$ or $\min(M_1 \triangle M_2) \preceq_{\triangle} M_2$. For example, we have that $\{5, 8\} \preceq_{\triangle} \{1, 8\}$ since the minimum element in $\{5, 8\} \triangle \{1, 8\} = \{1, 5\}$ is 1, which is in $\{1, 8\}$. Notice that the $\subseteq_{\triangle}$-relation is a refinement of the $\subseteq_{\triangle}$-relation in the sense that if $M_1 \subseteq_{\triangle} M_2$ then $M_1 \preceq_{\triangle} M_2$. This fol-
lows the intuition that the more elements a match has, the more consecutive it is. Moreover, one can prove that the \( \leq_{\text{next}} \) relation forms a total order among matches, implying the existence of a maximum over any finite set of matches.

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

We now define the semantics of the next selector \( \text{NXT}(\varphi) \): for any CEPL formula \( \varphi \) we have \( M \in \text{NXT}(\varphi) \) if \( M \in \varphi \) and for every match \( M' \) such that \( M \leq_{\text{next}} M' \) and \( \text{max}(M) = \text{max}(M') \), it holds that \( M' \notin \varphi \). In our running example, \{1,8\} satisfies \( \text{NXT}(\varphi_1) \) on \( S \), as there is no “more consecutive” match satisfying \( \varphi_1 \) in the same prefix. Note that we compare matches with respect to \( \leq_{\text{next}} \) that have the same final position. This ensures that the maximum match always exists and that the optimality of a match only depends on the matches over the same prefix.

Another selector that has been proposed in the literature is that of selecting only the maximal matches in terms of inclusion. This corresponds to obtaining those matches that are as informative as possible, and therefore contain the biggest sets of events. Formally, for any CEPL formula \( \varphi \) we have that \( M \in \text{MAX}(\varphi) \) \( (S, i, \nu) \) holds if \( M \in \varphi \) \( (S, i, \nu) \) and for all matches \( M' \) such that \( M \subseteq M' \) and \( \text{max}(M) = \text{max}(M') \), it holds that \( M' \notin \varphi \). Coming back to our example, the \( \text{MAX} \) selector will output both matches \{1,8\} and \{5,8\} for \( \varphi_1 \), given that both matches are maximal in terms of set inclusion. On the contrary, formula \( \varphi_3 \) produced matches \{3,6,7\}, \{3,4,7\}, and \{3,4,6,7\}. Then if we evaluate \( \text{MAX}(\varphi_3) \) over the same stream, we will obtain only \{3,4,6,7\} as output, which is the maximal match. It is interesting to note that if we evaluate \( \text{NXT}(\varphi_3) \) over the stream we will also get \{3,4,6,7\} as the only output, illustrating that next yields matches with maximal information.

So far we have extensively discussed the foundations of CEPL. We presented a formal language with well-defined semantics that contains most of the operators found in the literature, including the so-called selection strategies. This is an important and foundational first step, but is not enough for defining a complete and practical framework for CEP. In the rest of the paper we study several practical aspects of CEPL. We start by discussing the syntactic form of CEPL formulas, and define syntactic restrictions that characterize semantic properties of interest. Then, we present a computational model and show how CEPL formulas in the introduced syntactic fragments can be translated in this model. Moreover, we show how to evaluate this model efficiently. Finally, we put all pieces together and present a complete framework for evaluating the studied CEPL formulas in practice.

### 6. Syntactic Analysis of CEPL

In this section we study the syntactic form of CEPL formulas, and define the classes of well-formed and safe formulas. These classes are based on syntactic restrictions that characterize semantic properties of interest.

Then, we define a convenient normal form for CEPL and show that any formula can be rewritten in this form.

#### 6.1 Syntactic restrictions of formulas

The definition of CEPL provides well-defined semantics for all formulas, allowing for a more concise theoretical analysis. However, there are some formulas whose semantics can be unintuitive. Consider for example the formula:

\[
\varphi_5 = (H \text{ AS } x) \text{ FILTER } (y, \text{tmp} \leq 30).
\]

Here, \( x \) will be naturally bounded to the only element in a match, but \( y \) will not add a new position to a match. By the semantics of CEPL, a valuation \( \nu \) for \( \varphi_5 \) must assign a position for \( y \) that satisfies the filter, but such position is not restricted to occur in the match. Moreover, \( y \) is not necessarily bounded to any of the events seen up to the last element in the match, and thus a match could depend on future events. For example, if we evaluate \( \varphi_5 \) over our running example \( S \) (Figure 1), we have that \( \{2\} \in \varphi_5 \) \( (S) \), however, a streaming evaluation of \( \varphi_5 \) would have to wait until the event at position 6 to output this match.

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

\[
\begin{align*}
\text{bound}(R \text{ AS } x) &= \{x\} \\
\text{bound}(\rho \text{ FILTER } \alpha) &= \text{bound}(\rho) \\
\text{bound}(\rho_1 \text{ OR } \rho_2) &= \text{bound}(\rho_1) \cap \text{bound}(\rho_2) \\
\text{bound}(\rho_1 \text{ UNLESS } \rho_2) &= \text{bound}(\rho_1) \\
\text{bound}(\rho_1 \text{ OP } \rho_2) &= \text{bound}(\varphi_1) \cup \text{bound}(\varphi_2) \\
\text{bound}(\text{SEL}(\rho)) &= \text{bound}(\rho)
\end{align*}
\]

where \( \text{OP} \in \{\ldots, \text{AND}, \text{ALL}\} \) and \( \text{SEL} \) is any selection strategy. Note that for the \( \text{OR} \) operator a variable must be defined in both formulas in order to be bounded. Similarly, in the case of \( \text{UNLESS} \) the variables that count are the ones in \( \varphi_1 \) since \( \varphi_2 \) is just checking that some matches do not exist. We say that CEPL formula \( \varphi \) is well-formed if for every sub-formula of the form \( \rho \text{ FILTER } \alpha \) and every variable \( x \in \text{var}(\alpha) \), there is another sub-formula \( \rho_x \) such that \( x \in \text{bound}(\rho_x) \) and \( \rho \) is a sub-formula of \( \rho_x \). Note that this definition allows for including filters with variables defined in a wider scope. For example, formula \( \varphi_4 \) in Section 2 is well-formed although variable \( x \) is used in the filter \( y, \text{id} = x \text{id} \) and defined outside the \( \rho \)-operator.

As it was previously discussed, we would like to consider only CEPL formulas that can output matches as soon as the last position of the match is seen. We formalize this with the notion of streamable formulas. Given streams \( S_1 \) and \( S_2 \), we say that \( S_1 \) is equal to \( S_2 \) up to position \( i \), denoted by \( S_1 = S_2 \) if \( S_1[j] = S_2[j] \) for each \( j \leq i \). Then, we say that a formula \( \varphi \) is streamable if for every match \( M \) and stream \( S = t_0 t_1 \ldots \) it holds...
that $M \in \varphi(S)$ if, and only if, $M \in \varphi(S')$ for every stream $S'$ such that $S = \text{max}(M)$. In other words, streamable formulas can (in principle) be evaluated in a streaming fashion given that the belonging of $M$ to $\varphi(S)$ only depends on the prefix $t_0t_1...t_{\text{max}(M)}$.

The next result shows that if we restrict to the class of well-formed formulas, we do have the streamable property.

**Theorem 1.** Every well-formed formula is streamable.

One can easily argue that it would be desirable for a CEP-system 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 can easily produce unsatisfiable formulas. For example, the formula $\forall t. \text{AS} x$ is not satisfiable (i.e. $[\forall t. \varphi](S) = \emptyset$ for every $S$) because variable $x$ cannot be assigned to two different positions in the stream. This issue arises when variables are reused on conjunctive operators like sequencing (;) or ALL. On the other hand, we do not want to be too conservative and disallow the reuse of variables in the whole formula (otherwise formulas like $\varphi_2$ in Section 2 would not be permitted).

This motivates the notion of safe CEPL formulas. We say that a CEPL formula is safe if for every subformula of the form $\varphi_1 \text{OP} \varphi_2$ with OP $\in \{\land, \lor, \text{ALL}\}$ it holds that $\text{vdef}(\varphi_1) \cap \text{vdef}(\varphi_2) = \emptyset$. For example, all CEPL formulas introduced in this paper are safe except for $\psi$.

The safe notion is a mild but useful restriction to help the evaluation of CEPL and can effectively be checked during parsing time. However, safe formulas are a subset of CEPL and it could be the case that this prevents users from writing certain patterns. We show in the next result that this is never the case for the core fragment. Formally, we say that two CEPL formulas $\varphi$ and $\psi$ are equivalent, denoted by $\varphi \equiv \psi$, if for every stream $S$ and match $M$, it is the case that $M \in [\varphi](S)$ if, and only if, $M \in [\psi](S)$.

**Theorem 2.** Let $\varphi$ be a core-CEPL formula. Then, there is a safe formula $\varphi'$ such that $\varphi \equiv \varphi'$ and $|\varphi'|$ is at most exponential in $|\varphi|$.

By this result, for core-CEPL we can restrict our analysis to safe formulas without losing expressiveness of the language. Instead, if we do not impose the safe restriction, we will have to assume an exponential blow-up in the rewriting of formulas (see Section 4 for further discussion).

### 6.2 LP-normal form

Now we study how to rewrite CEPL formulas in order to simplify the evaluation of unary filters. Intuitively, filter operators in a CEPL formula can become difficult to handle for a CEP query engine. As it was previously motivated by formula $\varphi_1$ and $\varphi'_1$ in Section 2 it is easier for a query optimizer to evaluate formulas where unary predicates are applied directly over their variables (e.g. $\varphi'_1$) and not anywhere in the formula (e.g. $\varphi_1$). This motivates the definition of formulas in locally parametrized normal form (LP-normal form). Let $P(R)$ be the set of all predicates $P \in P(R)$ such that $\text{arity}(P) = 1$. Furthermore, define $F_n(R) \subseteq F(R)$ to be the set of all selection formulas constructed from atomic predicates in $P_n(R)$. Then we say that $\varphi$ is in LP-normal form if the following condition holds: for every sub-formula $\varphi'$ FILTER $\alpha$ of $\varphi$, if $\alpha$ contains at least one predicate in $P_n(R)$, then $\varphi' = R \text{AS} x$ for some $R$ and $x$, and $\alpha \in F_n(R)$ with $\text{var}(\alpha) = \{x\}$. In other words, all filters containing unary predicates are applied directly to the definitions of their variables. For instance, formula $\varphi'_1$ is in LP-normal form while formulas $\varphi_1$ and $\varphi_2$ are not. Note that non-unary predicates are not restricted, and they can be used anywhere in the formula.

One can easily see the advantage for the query engine of using only formulas in LP-normal form (see Section 4 for further discussion). However, formulas that are not in LP-normal form can still be very useful for declaring patterns. To illustrate this, consider the formula:

$$
\varphi_6 = (T \text{AS} x): (((T \text{AS} y \text{FILTER } x\.\text{temp} \geq 40) \lor (H \text{AS} y \text{FILTER } x\.\text{temp} < 40))
$$

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 kind of conditional statements can be very useful for users and a serious problem for query engines. Fortunately, the next result shows that one can always rewrite a formula to an equivalent LP-normal form formula with an exponential blow-up in the size of the formula.

**Theorem 3.** Let $\varphi$ be a CEPL formula. Then, there is a CEPL formula $\psi$ 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.

### 7. A Computational Model for CEP

In this section, we introduce a formal computational model for evaluating CEPL formulas called match automata. 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 \cite{32, 10}, and relational algebra and SQL \cite{7, 40}. Here, we
propose match automata as the intermediate evaluation model for CEPL and show later how to compile any (unary) CEPL formula into a match automaton.

As its name suggests, match automata (MA) are an extension of Finite State Automata (FSA). The first difference from FSA comes from handling streams instead of words. A match automaton 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 are infinitely many 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) [15]. 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. Match automata are extended analogously, which is reflected in transitions labeled by (unary) formulas over tuples.

The last difference addresses the need to output matches instead of a boolean answer. A well known extension for FSA are Finite State Transducers [18] which are automata capable of producing an output whenever an input element is read. We follow this idea: MA are allowed to generate and output matches when reading a stream, similar to the class of synchronized transducers [27] (i.e. transducers whose input and output have the same length). Note that, although general transducers have bad decidability properties [18], the class of synchronized transducers is closed under union, intersection, and complement, and most of their associated problems are decidable [27]. In particular, our model inherit the good properties of synchronized transducers which are exploited in Section 8 for building MA from CEPL formulas.

Before defining the MA model we need some basic definitions. Fix a schema $\mathcal{R}$ and let $F_u(\mathcal{R})$ be the set of all selection formulas with unary predicates (as defined in Section 6). Given $t \in \text{tuples}(\mathcal{R})$ and $\alpha \in F_u(\mathcal{R})$, we say that $t$ satisfies $\alpha$, denoted by $t \models \alpha$, if $\sigma_t(\alpha) = 1$ where $\sigma_t$ is the function that assigns $t$ to every variable in $\alpha$ (i.e. $\sigma_t(x) = t$ for every $x \in \text{var}(\alpha)$). Finally, without loss of generality we suppose that $F_u(\mathcal{R})$ contains predicates of the form $\text{"type"}(x) = R$ for every $R \in \mathcal{R}$. This will help the automata model to check whether a tuple is of type $R$ or not.

Let $\mathcal{R}$ be a schema and $\bullet, \circ$ be two symbols. A match automaton (MA) over $\mathcal{R}$ is a tuple $A = (Q, \Delta, I, F)$ where $Q$ is a finite set of states, $\Delta \subseteq Q \times (F_u(\mathcal{R}) \times \{\bullet, \circ\}) \times Q$ is the transition relation, and $I, F \subseteq Q$ are the set of initial and final states, respectively. Given an $\mathcal{R}$-stream $S = t_0 t_1 \ldots$, a run $\rho$ of $A$ over $S$ is a sequence of transitions: $\rho : q_0 \xrightarrow{a_0/m_0} q_1 \xrightarrow{a_1/m_1} \ldots \xrightarrow{a_n/m_n} q_{n+1}$ such that $q_0 \in I$, $t_i \models a_i$ and $(q_i, a_i, m_i, q_{i+1}) \in \Delta$ for every $i \leq n$. We say that $\rho$ is accepting if $q_{n+1} \in F$ and $m_n = \bullet$. Intuitively, the set of values $i$ such that $m_i = \bullet$ in a run represent the set of transitions generated by that run. It is then natural to ask for the last position to be in the match, as otherwise a match could depend on future events. We denote by $\text{Run}_n(A, S)$ the set of accepting runs of $A$ over $S$ of length $n$. Further, we denote by $\text{match}(\rho)$ the set of positions where the run marks the stream, namely $\text{match}(\rho) = \{i \in [0, n] \mid m_i = \bullet\}$. Given a stream $S$ and $n \in \mathbb{N}$, we define the set of matches of $A$ over $S$ at position $n$ as: $[A]_n(S) = \{\text{match}(\rho) \mid \rho \in \text{Run}_n(A, S)\}$ and the set of all matches as $[A](S) = \bigcup_n [A]_n(S)$. Although $[A](S)$ can be an infinite set of matches, $[A]_n(S)$ is always finite.

As an example, consider the match automaton $A$ depicted in Figure 3. In this MA, each transition $\alpha(x) \bullet$ marks one $H$-tuple and each transition $\beta(x) \circ$ marks a sequence of $T$-tuples with temperature bigger than 40. Note also that the transitions labeled by $\text{TRUE} \circ \bullet$ allows $A$ to arbitrarily skip any of the input tuples in the stream. Then, for every stream $S$, $[A](S)$ represents the set of all matches 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 match automata are designed to be an evaluation model for an expressive subfragment of CEPL, called unary CEPL (see Section 8 for the formal definition). Several computational models have been proposed for complex event processing [26, 39, 40, 43]; most of them are informal and complex extensions of finite state automata. In our framework, we want to give a step back compared to previous proposals and define a simple but powerful model that captures the regular core of CEPL. With “regular” we mean all CEPL formulas that can be checked with finite state memory. Intuitively, formulas like $\varphi_1, \varphi_2$ and $\varphi_3$ presented in Section 2 can be evaluated using a bounded amount of memory. In contrast, formula $\varphi_4$ needs unbounded memory to store candidate events seen in the past, and thus, it calls for a more sophisticated model (e.g. data automata [44]). Of course, one would like to have a full-fledged model for CEPL, but this is not possible if we do not understand first its regular core.

For these reasons, a computational model for the whole CEP logic is left for future work (see Section 10 for more discussion).

The MA model has good closure properties, for example, under union, intersection, complement and determinization. Formally, we say that a match automaton $A$ is deterministic if $|I| = 1$ and for any two transitions $(p, a_1, m_1, q_1)$ and $(p, a_2, m_2, q_2)$, either $a_1$ and $a_2$ are mutually exclusive (for every $t$ it is not true that $t \models a_1$ and $t \models a_2$), or $m_1 \neq m_2$ (see 42 for a similar definition of deterministic letter-to-letter transducers).

![Figure 3: A match automaton that can generate an unbounded amount of matches over a stream.](image-url)

Here $\alpha(x) := \text{type}(x) = H$ and $\beta(x) := \text{type}(x) = T \land x.\text{temp} > 40.$
Then we say that MA are closed under determinization (complement) if for every MA $A$, there is a deterministic MA $A^\text{det}$ (a MA $A^c$ resp.) such that for every stream $S$ we have $[A^\text{det}](S) = [A](S) \land [A^c](S) = \{ M \in 2^N \mid M \text{ is finite} \} \lor [A](S)$ resp.). Furthermore, we say that $A$ is closed under union (intersection) if for every MA $A_1$ and $A_2$, there exists a MA $A$ such that for every stream $S$ we have $[A](S) = [A_1](S) \cup [A_2](S) ([A](S) = [A_1](S) \cap [A_2](S)$ resp.).

**Proposition 1.** MA are closed under union, intersection, complement, and determinization.

Having a good computational model at hand, in the following two sections we show how to compile and efficiently evaluate the unary fragment of CEPL.

### 8. Compiling Unary CEPL

In this section we show how to compile a (well-formed, unary) CEPL formula $\varphi$ into an equivalent MA $A_{\varphi}$, meaning that $[\varphi](S) = [A_{\varphi}](S)$ for every stream $S$, to later evaluate $A_{\varphi}$ over streams. Formally, we say that a CEPL formula $\varphi$ is unary if for every subformula of $\varphi$ of the form $\alpha \text{ FILTER } \alpha$, all predicates of $\alpha$ are unary (i.e. $\alpha \in \mathcal{F}_u(R)$). For example, formulas $\varphi_1$, $\varphi_2$, and $\varphi_3$ in Section 2 but formula $\varphi_4$ is not (the predicate $y.id = x.id$ is binary). It is important to mention that, although the unary fragment seems restricted, we have shown in Section 3 that it already presents non-trivial computational challenges like, for example, in evaluating formula $\varphi$ which is a unary formula. The evaluation of full CEPL is an interesting problem which requires new insights on rewriting techniques and more powerful computational models featuring translations and efficient evaluation strategies; we envision this as future work.

Now we present the compilation of unary core-CEPL into MA. This construction is intimately related with the safeness condition and LP-normal form (see Section 6).

**Theorem 4.** For every well-formed formula $\varphi$ in unary core-CEPL, there exists a MA $A_{\varphi}$ equivalent to $\varphi$. Furthermore, $A_{\varphi}$ is of size at most linear in $|\varphi|$ if $\varphi$ is safe and in LP-normal form, and of size at most double exponential in $|\varphi|$ otherwise.

The proof of Theorem 4 goes by first converting $\varphi$ into an equivalent CEPL formula $\varphi'$ in LP-normal form (Theorem 3) and then building an equivalent MA from $\varphi'$. We show that there is an exponential blow-up for converting $\varphi$ into LP-normal form. Furthermore, we show that the output of the second step is of linear size if $\varphi'$ is safe, and of exponential size otherwise. Intuitively, this proves our claim from Section 3 as when we restrict the language to safe formulas, we can provide more efficient evaluation strategies for CEPL.

Next we focus on how to construct MA from formulas with extended operators like AND, ALL, and UNLESS. In contrast to the core fragment, these operators are more complicated to evaluate, which is reflected in the size of their respective MAs. Let $\varphi_1$ and $\varphi_2$ be two unary core-CEPL formulas and $X = \text{vdef}_*(\varphi_1) \cap \text{vdef}_*(\varphi_2)$.

**Theorem 5.** Let $\varphi = \varphi_1 \text{ OP } \varphi_2$ be a CEPL formula with OP $\in \{ \text{AND, ALL, UNLESS} \}$, and let $A_{\varphi_1}$ and $A_{\varphi_2}$ be two MA equivalent to $\varphi_1$ and $\varphi_2$, respectively. Then there is a MA $A_{\varphi}$ equivalent to $\varphi$ of size at most $O(|A_{\varphi_1}| \cdot |A_{\varphi_2}| \cdot 2^{|X|})$ if OP $\in \{ \text{AND, ALL} \}$ and at most $O(|A_{\varphi_1}| \cdot 2^{|A_{\varphi_2}|})$ if OP = UNLESS.

The previous result shows the cost that a CEP-system will have to incur in if these extended operators are used. Note that the quadratic or exponential cost is just with one extended operator, and this does not include the cost of bringing the formula into LP-normal form. Furthermore, the proof of this result again shows the advantage of using safe formulas: if $\varphi_1$ AND $\varphi_2$ or $\varphi_1$ ALL $\varphi_2$ are safe, then the cost $2^{|X|}$ in their constructions can be avoided.

We now study how to build MA for CEPL formulas with selection strategies. For this, we present our results using a more general framework, in which selection strategies are applied directly over MA. Let $A$ be a MA and SEL a selector in $\{ \text{STRICT, NXT, MAX} \}$. Then we say that a MA $A_{\text{SEL}}$ is equivalent to SEL$(A)$ whenever $\text{SEL}(A)(S) = [A_{\text{SEL}}](S)$ for every stream $S$.

**Theorem 6.** Let SEL be a selection strategy. For any MA $A$, there is an MA $A_{\text{SEL}}$ equivalent to SEL$(A)$. Furthermore, the size of $A_{\text{SEL}}$ is, w.r.t. the size of $A$, at most linear if SEL = STRICT, at most exponential if SEL = NXT and at most double exponential if SEL = MAX.

At first the above result might seem unintuitive, specially in the case of NXT and MAX. It is not immediate to show that there exists a MA that can keep an unbounded number of different matches with a finite number of states. However, this can be done with finite memory but with an exponential or double exponential (for the case of NXT and MAX, respectively) blow-up in the number of states.

Theorem 6 concludes our study of the compilation of unary CEPL into MA. We have shown that MA is a powerful model for evaluating CEPL formulas, and that can be further exploited to evaluate selection strategies. In the next section we focus on how to evaluate MA.

### 9. Evaluation of Unary CEPL

In this section, we put all pieces together and present a framework for efficiently evaluating unary CEPL formulas. We start by showing an efficient algorithm for evaluating every match automata over a stream in constant time per data item. Then by joining this with the previous results, we show a complete framework for efficient evaluation of unary CEPL formulas, taking care of all steps that can produce some additional cost in the process.

To define a notion of efficiency for our framework is challenging since we would like to compute matches in...
one pass and using a restricted amount of resources. Streaming algorithms [18, 20] are a natural example of this, 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., sublinear or logarithmic in \( n \)). However, as we want to output matches, we cannot expect to use less than linear space in the processed data, because a match could be as long as the stream itself. Another problem for defining the concept of efficiency is that the input object (a stream) is infinite. For this reason, we associate to a stream \( S \) a special instruction \( \text{yield}_S \) that returns the next element of \( S \). Then, given a function \( f \) we say that an \( f \)-evaluation strategy for a match automaton \( A \) is an algorithm such that for every stream \( S \) (1) the time spent between two calls to \( \text{yield}_S \) is bounded by \( O(f(|A|) \cdot |t|) \), where \( t \) is the tuple returned by the first of such calls, and (2) a data structure \( D \) is maintained in memory, such that after calling \( \text{yield}_S \) \( n \) times, the set of matches \( |A|_n(S) \) can be enumerated from \( D \) with constant delay. The latter condition basically imposes that no processing is done during output generation. Formally, it requires the existence of a routine \( \text{enumerate} \) that receives \( D \) as input and outputs all matches in \( |A|_n(S) \) without repetitions, while spending a constant amount of time before and after each output. Moreover, when the algorithm outputs a match \( M \), the time to generate this output can only be linear in \( M \). We remark that the requirement (1) is a natural restriction imposed in the streaming literature [33], while (2) is the minimum that we can ask if an arbitrarily large set of arbitrarily large outputs must be produced [16]. Finally, we say that there exists an efficient evaluation strategy for a match automaton \( A \) if there exists an \( f \)-evaluation strategy for some function \( f \).

The notion of efficient evaluation strategy introduced here considers the data complexity of the problem, namely, the number of states and transitions of \( A \) are assumed constant and, thus, the function \( f \) is not important in the asymptotic analysis. Instead, the notion of \( f \)-evaluation strategy is more fine grained making explicit the dependency between the size of the match automaton and the update of a tuple. Notice that if the schema and automata are fixed and the values use a fixed amount of memory, we can process each tuple in constant time. Having a good notion of efficiency, we proceed to show what is probably the most important result of this paper, specially from a practical point of view.

**Theorem 7.** For every deterministic MA \( A \), there exists an \( f \)-evaluation strategy with \( f = |A| \).

Note that, by Proposition 1, we know that every MA can be determined and therefore the previous results implies an efficient evaluation strategy in terms of data complexity (in this case \( f \) is going to be an exponential function).

To avoid this exponential factor, another possibility for optimizing the evaluation of MA is to restrict the set of results; as discussed in Section 5 this has been the main motivation in the literature to introduce the \( \text{NXT} \) selection strategy. Indeed, in the following results we show that, if the \( \text{NXT} \) selection strategy is used to restrict the output of matches, the exponential factor of the evaluation strategy of any match automata can be avoided.

**Theorem 8.** For every MA \( A \), there is an \( f \)-evaluation strategy for computing \( \text{NXT}(A) \), where \( f = |A| \).

It is important to say that the evaluation strategy is linear in the size of \( A \) avoiding constructing the equivalent match automata for \( \text{NXT}(A) \) (see Theorem 6). In particular, Theorem 8 is avoiding two exponential blow-ups: one for constructing the equivalent automata \( A_{\text{NXT}} \) for \( \text{NXT}(A) \) and the other for determining \( A_{\text{NXT}} \).

Now, we have all the pieces to show how to evaluate any unary CEPL formula (even with selection policies) efficiently. In Figure 4 we show the evaluation cycle of a CEPL formula in our framework with the main results of the paper that are needed for the full process. For understanding the evaluation cycle, consider a unary CEPL formula \( \varphi \) (probably with extended operators or selection strategies). The processing of \( \varphi \) starts in the Parser module, where we check if \( \varphi \) is well-formed (WF) and safe. These conditions are important to ensure that \( \varphi \) is streamable (Theorem 1) and satisfiable. Although unsafe formulas are not necessarily unsatisfiable, if a CEP system wants to allow unsafe formulas, it will have to assume an exponential blow-up in rewriting \( \varphi \) into its safe version (Theorem 2).

The next module (Query Rewrite) rewrites a well-formed and safe formula \( \varphi \) into a formula \( \varphi' \) in LP-normal form. For transforming CEPL formulas into LP-normal form, one can use the rewriting process of Theorem 8 which, in the worst case, can produce an exponential blow-up in the size of \( \varphi' \). To avoid this cost, in many cases one can apply local rewriting rules which has been extensively studied in relational database management systems (DBMS) [7, 10]. For example, formula \( \varphi_1 \) in Section 2 is converted into \( \varphi_1' \) by applying a filter push on \( (x.tmp > 40 \land x.id = 0) \) and \( (y.hum <= 25 \land y.id = 0) \), avoiding the exponential blow-up of Theorem 3. As in DBMS, this approach can produce formulas in LP-normal form of polynomial size (w.r.t. \( \varphi \)). Unfortunately, we cannot apply this technique over formulas like \( \varphi_6 \) in Section 6 maintaining the blow-up of Theorem 3. Despite this, formulas like \( \varphi_6 \) are rather uncommon in practice, and therefore one can assume that local rewriting rules will usually produce LP-formulas of polynomial size.

The third component (Compilation) receives formula \( \varphi' \) in LP-normal form and builds a match automaton \( A_\varphi \). By Theorem 4 we can construct \( A_\varphi \) in polynomial time in the size of \( \varphi' \) whenever \( \varphi \) is a core-CEPL formula. On the contrary, if \( \varphi \) has extended operators or selection strategies, one has to afford an exponential blow-up with respect to the number of AND, ALL and UNLESS in \( \varphi \). When \( \text{NXT} \) is used or double-exponential blow-up whenever \( \text{MAX} \) is used (Theorem 6). As the number of extended operators or selection policies is rather low in
CEPL → \textbf{Parser (Th. 1, 2)} \quad \downarrow \quad \text{WF and safe} \\
\textbf{Query Rewrite (Th. 3)} \quad \downarrow \quad \text{LP-normal form} \\
\textbf{Compilation (Th. 4, 5, 6)} \quad \downarrow \quad \text{Match automaton} \\
Stream → \textbf{Evaluation (Th. 7, 8)} \quad \downarrow \quad \text{output (complex events)}

Figure 4: Evaluation framework for CEPL.

practice, the cost of compiling them should not affect the overall performance significantly.

The last module (Evaluation) takes the MA \( \mathcal{A}_\varphi \) produced by the Compilation module and evaluates it by using the efficient evaluation strategy presented in Theorem 7. For this, the efficient evaluation strategy requires the determination of \( \mathcal{A}_\varphi \) which requires another exponential blow-up in the process. Note that, in terms of data complexity, this blow-up is small compared to the stream and only affects the constant-time needed for each new event. To avoid this determination procedure, we can use the \( \text{NXT} \) selection strategy to filter the output of \( \mathcal{A}_\varphi \) and exploit the linear selection strategy presented in Theorem 8.

\textbf{THEOREM 9.} There exists an efficient evaluation strategy for every unary CEPL formula.

Summing up, our framework can process any unary CEPL formula \( \varphi \) efficiently, with time per item proportional to \( \varphi \) if the Query rewrite and Compilation module do not increase the size of \( \varphi' \) and \( \mathcal{A}_\varphi \) significantly.

10. CONCLUSIONS AND FUTURE WORK

We have presented a formal framework for Complex Event Processing. We studied and formalized the different operators found in the literature and the so-called selection strategies, and we introduced a logic called CEPL that captures the main features of CEP. Towards building a framework for evaluating this language, we provided many interesting concepts and results for CEPL like syntactic restrictions (well-formed and safe), the LP-normal form, an evaluation model (MA), translation from unary CEPL to MA, and efficient evaluation of MA, among others. By gathering all these results together, we proposed a formal and practical framework for efficiently evaluating unary CEPL.

This paper settles the basic foundations for CEP, stimulating many further research directions. In particular, a natural next step is the study of the evaluation of non-unary CEPL formulas, which require new insight in the rewriting of formulas and new computational models. Furthermore, a relevant problem for the area is to provide efficient evaluation strategies for these new computational models. Another problem in this line is the design of new selection strategies. In Section 5 we introduce three important selection strategies but one can envision many other useful strategies that could boost the evaluation of queries.

Finally, we have studied the fundamental features of CEP languages, leaving other features outside in order to keep the language and analysis simple. These features include time windows, aggregation, consumption policies, among others (see [25] for a more exhaustive list). We believe that each of these features can be used to extend CEPL in new directions to establish more complete frameworks for CEP.

11. REFERENCES

[1] Esper enterprise edition website.
http://www.espertech.com/. Accessed: 2017-06-09.
[2] Oracle stream analytics website.
http://www.oracle.com/technetwork/middleware/complex-event-processing/documentation/index.html.
Accessed: 2017-06-09.
[3] Real time intelligence and complex event processing.
http://www.complexevents.com/. Accessed: 2017-06-09.
[4] Sase website. http://sase.cs.umass.edu/. Accessed: 2017-06-09.
[5] T-rex website. https://github.com/deib-polimi/TRex.
Accessed: 2017-06-09.
[6] 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.
[7] S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases: the logical level. Addison-Wesley, 1995.
[8] A. Adi and O. Etzion. Amit-the situation manager. VLDB Journal, 2004.
[9] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In SIGMOD, 2008.
[10] A. V. Aho. Algorithms for finding patterns in strings. In Handbook of Theoretical Computer Science. 1990.
[11] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. Proceedings of the VLDB Endowment, 2008.
[12] R Alur and D. L. Dill. A theory of timed automata. Theoretical computer science, 1994.
[13] 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.
[14] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenberg, and J. Widom. Stream: The stanford stream data manager (demonstration description). In SIGMOD, 2003.
[15] A. Arasu, S. Babu, and J. Widom. The cq1 continuous query language: Semantic foundations and query execution. The VLDB Journal, 2006.
[16] G. Bagdon, M. M. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Proc. CSL, 2007.
[17] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In CIDR, 2007.
APPENDIX

A. PROOFS OF SECTION 5

A.1 Proof of Lemma 1

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

Anti-symmetric. Consider any two matches $M_1$ and $M_2$ such that $M_1 \leq_{next} M_2$ and $M_2 \leq_{next} M_1$. $M_2 \leq_{next} M_1$ means that either $M_1 = M_2$ or (1) $\min\{(M_1 \cup M_2) - (M_1 \cap M_2)\} \in M_1$, and $M_1 \leq_{next} M_2$ that either $M_2 = M_1$ or (2) $\min\{(M_2 \cup M_1) - (M_2 \cap M_1)\} \in M_2$. If (1) were true, it would mean that (2) could not be true, so $M_2 = M_1$ would have to be true, becoming a contradiction. So, the only possible scenario is that $M_1 = M_2$.

Transitivity. Consider any three matches $M_1$, $M_2$, and $M_3$ such that $M_1 \leq_{next} M_2$ and $M_2 \leq_{next} M_3$. $M_1 \leq_{next} M_2$ means that either $M_1 = M_2$ or (1) $\min\{(M_1 \cup M_2) - (M_1 \cap M_2)\} \in M_2$. If $M_2 = M_3$, then $M_1 \leq_{next} M_3$ because $M_2 \leq_{next} M_3$. Now, if $M_1 \neq M_2$, then (1) must hold, which means that the lowest element that is either in $M_1$ or $M_2$, but not in both, has to be in $M_2$. Let’s call this element $l_1$. $M_2 \leq_{next} M_3$ means that either $M_2 = M_3$ or (2) $\min\{(M_2 \cup M_3) - (M_2 \cap M_3)\} \in M_3$. Again, if $M_2 = M_3$, then $M_1 \leq_{next} M_3$ because $M_1 \leq_{next} M_2$. Now, if $M_2 \neq M_3$, then (2) must hold, which means that the lowest element that is either in $M_2$ or $M_3$, but not in both, has to be in $M_3$. Let’s call this element $l_2$.

Given that $M_1 = M_2$ and $M_2 \neq M_3$, define for every $i \in \{1, 2, 3\}$ and $j \in \{1, 2\}$ the set $M_i^{cl_j}$ as the set of elements of $M_i$ which are lower than $l_j$, i.e., $M_i^{cl_j} = \{x | x \in M_i \wedge x < l_j\}$. It is clear that $M_1^{cl_1} = M_2^{cl_1}$ and $M_1^{cl_2} = M_2^{cl_2}$, because of (1) and (2), respectively. Also, because of (2) it holds that $l_2 \notin M_2$, so $l_1 \neq l_2$.

Consider first the case where $l_1 < l_2$. This means that (3) $M_1^{cl_1} = M_3^{cl_1}$. Moreover, if $l_1$ were not in $M_3$, it would contradict (2), so (4) $l_1 \in M_3$ must hold. With (3) and (4), it follows that $l_1$ is the lowest element that is either in $M_1$ or $M_3$ but not in both, and it is in $M_3$. This proves that $\min\{(M_1 \cup M_3) - (M_1 \cap M_3)\} \in M_3$, and thus $M_1 \leq_{next} M_3$.

Now consider the case where $l_2 < l_1$. This means that (5) $M_1^{cl_2} = M_3^{cl_2}$. Because $l_2$ is not in $M_2$, it cannot be in $M_1$, otherwise it would contradict (1), so (6) $l_2 \notin M_1$ must hold. Also, because of (2) we know that (7) $l_2 \in M_3$ must hold. With (5), (6), and (7), it follows that $l_2$ is the lowest element that is either in $M_1$ or $M_3$ but not in both, and it is in $M_3$. This proves that $\min\{(M_1 \cup M_3) - (M_1 \cap M_3)\} \in M_3$, and thus $M_1 \leq_{next} M_3$.

Total. Consider any two matches $M_1$ and $M_2$. If $M_1 = M_2$, then $M_1 \leq_{next} M_2$ holds. Consider now the case where $M_1 \neq M_2$. Define the set $M = (M_1 \cup M_2) \setminus (M_1 \cap M_2)$ which is the set of all elements either in $M_1$ or $M_2$, but not in both. Because $M_1 \leq_{next} M_2$, there must be at least one element in $M$. In particular, this implies that there is a minimum element $l$ in $M$. If $l$ is in $M_2$, then $M_1 \leq_{next} M_2$ holds, and if $l$ is in $M_1$, then $M_2 \leq_{next} M_1$ holds.

B. PROOFS OF SECTION 6

B.1 Proof of Theorem 1

Let $\varphi$ be a well-formed formula. In order to prove that $\varphi$ is streamable we first define the following lemmas:

**Lemma 2.** Consider any CEPL formula $\varphi$, stream $S$, match $M$, valuation $\nu$, and $i \in \mathbb{N}$. If $M \in \llbracket \varphi \rrbracket (S, i, \nu)$, then $\nu(x) \in M$ for every $x \in \text{bound}(\varphi)$.

**Proof.** We prove this by induction over the formula $\varphi$:
- Consider $\varphi = R \mathrel{\text{AS}} x$. Then, $\text{bound}(\varphi) = x$ and, by definition, $M \in \llbracket \varphi \rrbracket (S, i, \nu)$ implies that $M = \{\nu(x)\}$. Therefore, the lemma holds.
- Consider $\varphi = \rho \mathrel{\text{FILTER}} \alpha$. Then, $M \in \llbracket \varphi \rrbracket (S, i, \nu)$ implies that $M \in \llbracket \rho \rrbracket (S, i, \nu)$, therefore by induction hypothesis $\nu(x) \in M$ for every $x \in \text{bound}(\rho)$. Moreover $\text{bound}(\varphi) = \text{bound}(\rho)$, thus $\nu(x) \in M$ for every $x \in \text{bound}(\varphi)$.
- Consider $\varphi = \rho_1 \mathrel{\text{OR}} \rho_2$. Then, $M \in \llbracket \varphi \rrbracket (S, i, \nu)$ implies that either $M \in \llbracket \rho_1 \rrbracket (S, i, \nu)$ or $M \in \llbracket \rho_2 \rrbracket (S, i, \nu)$. Without loss of generality, assume that it is the first case. Then, by induction hypothesis $\nu(x) \in M$ for every $x \in \text{bound}(\rho_1)$. Moreover, because $\text{bound}(\varphi) = \text{bound}(\rho_1 \cup \rho_2)$, then $\nu(x) \in M$ for every $x \in \text{bound}(\varphi)$.
- Consider $\varphi = \rho_1 \mathrel{\text{AND}} \rho_2$. Then, $M \in \llbracket \varphi \rrbracket (S, i, \nu)$ implies that there exist matches $M_1$ and $M_2$ with $M = M_1 \cdot M_2$ such that $M_1 \in \llbracket \rho_1 \rrbracket (S, i, \nu)$ and $M_2 \in \llbracket \rho_2 \rrbracket (S, \max(M_1) + 1, \nu)$. By induction hypothesis $\nu(x) \in M_i$ for every $x \in \text{bound}(\rho_i)$. Because $\text{bound}(\varphi) = \text{bound}(\rho_1) \cup \text{bound}(\rho_2)$ and both $M_1, M_2 \in M$, it holds that $\nu(x) \in M$ for every $x \in \text{bound}(\varphi)$.
- Consider $\varphi = \rho^+$. By definition, $\text{bound}(\varphi) = \emptyset$, therefore the lemma trivially holds.
- Consider $\varphi = \rho_1 \mathrel{\text{AND}} \rho_2$. Then, by definition $M \in \llbracket \varphi \rrbracket (S, i, \nu)$ means that $M \in \llbracket \rho_1 \rrbracket (S, i, \nu)$ and $M \in \llbracket \rho_2 \rrbracket (S, i, \nu)$. Therefore, by induction hypothesis $\nu(x) \in M$ for every $x \in \text{bound}(\rho_1) \cup \text{bound}(\rho_2) = \text{bound}(\varphi)$. 

Given that the lemma holds in all cases, the lemma is shown.

For the next lemma, define the set unbound($\varphi$) $\subseteq X$ such that $x \in \text{unbound}(\varphi)$ if there exist a sub-formula of the form $\varphi' \text{ FILTER } \alpha$, $x \in \text{var} (\alpha)$ and there does not exist another sub-formula $\varphi_x$ such that $x \in \text{bound} (\varphi_x)$ and $\varphi'$ is a sub-formula of $\varphi_x$. In other words, we define the set unbound($\varphi$) of all variables that witness that a formula is not well-formed.

**Lemma 3.** Consider any CEPL formula $\varphi$, stream $S$, match $M$, valuation $\nu$, and $i,j \in \mathbb{N}$. If $M \in [\varphi](S,i,\nu)$, $m \leq j$ for every $m \in M$ and $\nu(x) \leq j$ for all $x \in \text{unbound}(\varphi)$, then $M \in [\varphi](S',i,\nu)$ for every stream $S'$ such that $S \ann S'$.

**Proof.** We prove this by induction over $\varphi$:

- Consider $\varphi = R \text{ AS } x$. Then, by definition $M \in [\varphi](S,i,\nu)$ means that $M = \{x\}$, $\text{type}(S[x]) = R$ and $i \leq \nu(x)$. Moreover, because $\nu(x) \leq j$ then $S'$[$x$] = $S$[$x$], thus type($S$[$x$]) = $R$ and $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho \text{ FILTER } \alpha$. Then, $M \in [\varphi](S,i,\nu)$ implies that $M \in [\rho](S,i,\nu)$, and unbound($\varphi$) = unbound($\rho$), thus $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho)$. Consider any stream $S'$ such that $S \ann S'$. Then, by induction hypothesis $M \in [\rho](S',i,\nu)$, thus $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho_1 \text{ OR } \rho_2$. Then, $M \in [\varphi](S,i,\nu)$ means that either $M \in [\rho_1](S,i,\nu)$ or $M \in [\rho_2](S,i,\nu)$. Without loss of generality, assume that $\text{case 1}$. Because unbound($\rho_1$) $\subseteq$ unbound($\varphi$), $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_1)$. Consider any stream $S'$ such that $S \ann S'$. Then, by induction hypothesis $M \in [\rho_1](S',i,\nu)$, thus $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho_1 : \rho_2$. Then, $M \in [\varphi](S,i,\nu)$ implies that there exist matches $M_1$ and $M_2$ with $M = M_1 \cdot M_2$ such that $M_1 \in [\rho_1](S,i,\nu)$ and $M_2 \in [\rho_2](S,\max(M_1) + 1,\nu)$. Moreover, because $\nu(x) \leq j$ for all $x \in \text{unbound} (\varphi)$ then $\nu(x_1) \leq j$ for all $x_1 \in \text{unbound} (\rho_1) \cap \text{bound} (\rho_2)$. Also, if $x \in \text{unbound} (\rho_1) \cap \text{bound} (\rho_2)$, then by Lemma 2 it holds that $x \in M_2$. Moreover, $m \leq j$ for every $m \in M$ and $M_2 \subseteq M$, therefore $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_1)$, and similarly $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_2)$. Consider any stream $S'$ such that $S \ann S'$. Then, by induction hypothesis $M_1 \in [\rho_1](S',i,\nu)$ and $M_2 \in [\rho_2](S',\max(M_1) + 1,\nu)$, thus $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho_1 \text{ AND } \rho_2$. Then, by definition $M \in [\varphi](S,i,\nu)$ means that $M \in [\rho_1](S,i,\nu)$ and $M \in [\rho_2](S,i,\nu)$. Similarly to the case, by Lemma 2 it holds that $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_1) \cup \text{unbound} (\rho_1)$. Consider any stream $S'$ such that $S \ann S'$. By induction hypothesis $M \in [\rho_1](S',i,\nu)$ and $M \in [\rho_2](S',i,\nu)$, thus $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho_1 \text{ ALL } \rho_2$. By definition, $M \in [\varphi](S,i,\nu)$ means that there exist matches $M_1$ and $M_2$ such that $M = M_1 \cup M_2$ and both $M \in [\rho_1](S,i,\nu)$ and $M \in [\rho_2](S,i,\nu)$ hold. Again, similarly to the case it holds by Lemma 2 that $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_1) \cup \text{unbound} (\rho_1)$. Consider any stream $S'$ such that $S \ann S'$. Therefore, by induction hypothesis $M_1 \in [\rho_1](S',i,\nu)$ and $M_2 \in [\rho_2](S',i,\nu)$, thus $M \in [\varphi](S',i,\nu)$.
- Consider $\varphi = \rho_1 \text{ UNLESS } \rho_2$. By definition, $M \in [\varphi](S,i,\nu)$ implies that $M \in [\rho_1](S,i,\nu)$ and for all $M'$ and $\nu'$ such that $\min(M) \leq \min(M')$ and $\max(M') \leq \max(M)$, $M' \in [\rho_2](S,i,\nu' \text{ vdef, } (\rho_2))$. Moreover, unbound($\rho_1$) $\subseteq$ unbound($\varphi$) and unbound($\rho_2$) $\subseteq$ unbound($\varphi$) $\cup$ bound($\rho_1$), therefore $\nu(x) \leq j$ for all $x \in \text{unbound} (\rho_1) \cup \text{unbound} (\rho_2)$. Consider any stream $S'$ such that $S \ann S'$. Then, by induction hypothesis $M \in [\rho_1](S',i,\nu)$ and $M' \in [\rho_2](S',i,\nu')$ by induction hypothesis it would mean that $M' \in [\rho_1](S,i,\nu' \text{ vdef, } (\rho_2))$, and therefore there exist no such $M'$ and $\nu'$. Then, it holds that $M \in [\varphi](S',i,\nu)$.

Given that the lemma holds in all cases, the lemma is shown.

Now, with Lemma 3 the proof is straightforward: because $\varphi$ is well-formed then unbound($\varphi$) $= \emptyset$, and because of Lemma 3 (with $i = 0$ and $j = \max(M)$), for every match $M$ and stream $S$ it holds that $M \in [\varphi](S)$ if, and only if, $M \in [\varphi](S')$ for every stream $S'$ such that $S = \max(M) S'$, thus $\varphi$ is streamable.
B.2 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-CEPL formula. Formally, we say that a CEPL formula \( \varphi \) is in disjunctive-normal form if \( \varphi = (\varphi_1 \text{OR} \cdots \text{OR} \varphi_n) \), where for each \( i \in \{1, \ldots, n\} \), it is the case that:

- Every OR operator in \( \varphi_i \) occurs in the scope of a + operator.
- For every subformula of \( \varphi_i \) of the form \((\varphi'_i)^+\), it is the case that \( \varphi'_i \) is in disjunctive normal form.

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

**Lemma 4.** Every formula \( \varphi \) in core-CEPL can be translated into disjunctive-normal form in time at most exponential \(|\varphi|\).

**Proof.** We proceed by induction over the structure of \( \varphi \).

- If \( \varphi = R \text{ AS } x \), then \( \varphi \) is already free of OR.
- If \( \varphi = \varphi_1 \text{OR} \varphi_2 \), the result readily follows from the induction hypothesis.
- If \( \varphi = (\varphi')^+ \), by induction hypothesis \( \varphi \) can be translated into disjunctive normal form.
- If \( \varphi = \varphi' \text{FILTER} \alpha \), we know by induction hypothesis that \( \varphi' \) is equivalent to a formula \((\varphi_1 \text{OR} \cdots \text{OR} \varphi_m)\). Therefore, \( \alpha \) is equivalent to \((\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha \). We show that this latter formula is equivalent to \((\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha \). Let \( S \) be a stream and assume \( M \in \{[(\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha]\}(S) \). Then, there is some \( \nu \) such that \( M \in \{[(\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha]\}(S,0,\nu) \) and \( \nu S \equiv \alpha \). By definition of \( \text{FILTER} \), this implies that there is an \( i \in \{1, \ldots, n\} \) such that \( M \in \{[(\varphi_i \text{FILTER} \alpha)]\}(S,0,\nu) \). We can then immediately conclude that \( M \in \{[(\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha]\}(S,0,\nu) \), and thus \( M \in \{[(\varphi_1 \text{FILTER} \alpha) \cdots \text{FILTER} \alpha]\}(S) \).

The converse follows from an analogous argument.

- If \( \varphi = (\varphi_1; \varphi_2) \), by induction hypothesis we know that \( \varphi_1 \) is equivalent to a formula \((\varphi'_1 \text{OR} \cdots \text{OR} \varphi'_m)\) and \( \varphi_2 \) is equivalent to a formula \((\varphi'_2 \text{OR} \cdots \text{OR} \varphi'_m)\). Let \( \varphi' \) be defined by

\[
\varphi' = (\varphi'_1; \varphi'_2) \text{OR} \cdots \text{OR} (\varphi'_1; \varphi'_2) \text{OR} \cdots \text{OR} (\varphi'_1; \varphi'_2) \text{OR} \cdots \text{OR} (\varphi'_1; \varphi'_2) \text{OR} \cdots \text{OR} (\varphi'_1; \varphi'_2).
\]

We show that \( \varphi \equiv \varphi' \). Let \( S \) be a stream and let \( M \) be a match. If \( M \in \{[\varphi]\}(S) \), then there is a valuation \( \nu \) and two matches \( M_1 \) and \( M_2 \) such that \( M = M_1 \cdot M_2 \). We can then immediately conclude that \( M \in \{[\varphi]\}(S,0,\nu) \) and \( \nu S \equiv \alpha \). By definition of \( \text{FILTER} \), this implies that there is an \( i \in \{1, \ldots, n\} \) such that \( M \in \{[(\varphi_i \text{FILTER} \alpha)]\}(S,0,\nu) \). As \( M = M_1 \cdot M_2 \), it immediately follows that \( M \in \{[(\varphi_i \text{FILTER} \alpha)]\}(S,0,\nu) \), and thus \( M \in \{[\varphi]\}(S) \).

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

**Lemma 5.** Let \( \varphi \) be a core-CEPL formula in which every OR occurs inside the scope of a + operator, and let \( x \in \text{vdef}_+(\varphi) \). Then, for every match \( M \), valuation \( \nu \), stream \( S \) and \( i \in \mathbb{N} \) such that \( M \in \{[\varphi]\}(S,i,\nu) \), it is the case that \( x \in \text{dom}(\nu) \) and \( \nu(x) \in M \).

**Proof.** We proceed by induction on the structure of \( \varphi \). Let \( \nu \) be a valuation, \( S \) a stream, \( i \in \mathbb{N} \) and \( M \) a match.

- Assume \( \varphi = R \text{ AS } x \) and that \( M \in \{[\varphi]\}(S,i,\nu) \). By definition, we have \( M = \{\nu(x)\} \).
- Assume \( \varphi = \varphi' \text{FILTER} \alpha \) and that \( M \in \{[\varphi]\}(S,i,\nu) \). Let \( x \in \text{vdef}_+(\varphi') \). By definition, we have that \( M \in \{[\varphi']\}(S,i,\nu) \).
- If \( \varphi = (\varphi')^+ \), the condition trivially holds as \( \text{vdef}_+(\varphi') = \emptyset \).
- 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 \( M \in \{[\varphi]\}(S,i,\nu) \), then \( M = M_1 \cdot M_2 \), where \( M_1 \in \{[\varphi]\}(S,i,\nu) \). As \( x \in \text{vdef}_+(\varphi_1) \), by induction hypothesis we have that \( x \in \text{dom}(\nu) \) and \( \nu(x) \in M \).

**Lemma 6.** Let \( \varphi \) be a core-CEPL formula in which every OR occurs inside the scope of a + operator, and let \( S \) be a stream. If \( \varphi \) has a subformula \( \varphi' \) that is not under the scope of a + operator such that \( \{[\varphi']\}(S) = \emptyset \), then \( \{[\varphi]\}(S) = \emptyset \).

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

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

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

Lemma 7. Let $\varphi$ be a core-CEPL formula in disjunctive-normal form. Then $\varphi$ can be translated in linear time into a safe core-CEPL formula $\varphi'$.

Proof. Assume that $\varphi = \varphi_1 \text{ OR } \cdots \text{ OR } \varphi_n$ is a core-CEPL formula in disjunctive-normal form. By induction, we assume that every sub-formula of the form $(\varphi')+$ is already safe. Now we show that every unsafe $\varphi_i$ is unsatisfiable, and therefore it can be safely removed from the disjunction. Proceed by contradiction and assume $\varphi_i$ 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 $\text{vdef}_\alpha(\psi_1) \cap \text{vdef}_\alpha(\psi_2) \neq \emptyset$. Let $\alpha \in \text{vdef}_\alpha(\psi_1) \cap \text{vdef}_\alpha(\psi_2)$. By Lemma 8 we know that $\psi_1; \psi_2$ must be satisfiable. Therefore, there is a stream $S$, a valuation $\nu$ and a mapping $M$ such that $M \in \langle \psi_1; \psi_2 \rangle(S, 0, \nu)$. This implies the existence of two matches $M_1$ and $M_2$ such that $M_1 \in \langle \psi_1 \rangle(S, 0, \nu)$ and $M_2 \in \langle \psi_2 \rangle(S, 0, \nu)$. Since $x \in \text{vdef}_\alpha(\psi_i)$ and $\psi_i$ can only mention OR inside a + operator, by Lemma 8 we obtain that $\nu(x) \in M_1$. Similarly, as $x \in \text{vdef}_\alpha(\psi_2)$, we have $\nu(x) \in M_2$. But as $M = M_1 \cdot M_2$, we have that $M_1 \cap M_2 = \emptyset$, contradicting the facts that $\nu(x) \in M_1$ and $\nu(x) \in M_2$.

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 $\varphi$. 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 3 and 7. Indeed, given a core-CEPL formula $\varphi$, one can construct in exponential time an equivalent core-CEPL formula $\varphi'$ in disjunctive normal form. Then, from $\varphi'$ one can construct in linear time a safe formula in core-CEPL $\psi$ that is equivalent to $\varphi$, which is exactly what we wanted to show.

B.3 Proof of Theorem 3

First we give some definitions to simplify notation. Consider a formula $\alpha$ that has only unary predicates and such that negations are only used over predicates. From now on we consider all formulas to be in this form, since every formula can be written this way by pushing negations inside of the formula and changing $\lor$'s with $\land$'s and vice versa. We will also refer to $\alpha$ as the set literals that appear in $\alpha$, namely, the set of atomic formula or its negation (for consistency, if $p(x)$ appears in $\alpha$ only as $\neg p(x)$, we do not consider $p(x)$). Also, 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. We use the notation of $\alpha$ as formula and set indistinctly whenever its meaning is clear from the context. Given a CEPL formula of the form $\varphi = \varphi' \text{ FILTER } \alpha$, we define the set of unbound predicates of $\varphi$, written as $\text{unbound}_p(\varphi)$, as all the predicates (and negations) of the filters that are not instantiated, i.e., $p(x) \in \text{unbound}_p(\varphi)$ if $p(x) \in \alpha$ and $x \notin \text{bound}(\varphi')$. Notice that, as expected, if $\varphi$ is well-formed then $\text{unbound}_p(\varphi) = \emptyset$, but this does not apply to subformulas, i.e., there could be a subformula $\varphi'$ of $\varphi$ such that $\text{unbound}_p(\varphi') \neq \emptyset$.

Consider a well-formed CEPL formula with unary predicates. We first provide a construction for a CEPL formula in LP normal form and then prove that it is equivalent to $\varphi$. The first step of the construction is focused on rewriting the formula in a way that for every subformula $\varphi'$ it holds that $\text{unbound}_p(\varphi') = \emptyset$. The construction we provide to achieve this is the following. For every subformula of the form $\varphi' \text{ FILTER } \alpha$ and every predicate $p(x) \in \text{unbound}_p(\varphi')$, let $\varphi_x$ be the lowest subformula of $\varphi$ where $x$ is defined and that has $\varphi'$ as a subformula. Here we use the fact that $\varphi$ is well-formed, which means that such $\varphi_x$ must exist. Then, we rewrite the subformula $\varphi_x$ inside $\varphi$ as $\varphi'_x \text{ FILTER } p(x) \text{ OR } \varphi''_x \text{ FILTER } \neg p(x)$, where $\varphi'_x$, $\varphi''_x$, $\varphi'_x$, and $\varphi''_x$ are the same as $\varphi_x$ but replacing $p(x)$ with TRUE and FALSE, respectively.

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. A first approach is to take every predicate $p(x)$ that appears and move it down to every place where it was defined, i.e., to every subformula of the form $R \text{ AS } x$. The problem with this is that it would be forcing $p(x)$ to be true, even though this might not be necessary, for example if $p(x)$ appears in one side of a propositional disjunction. To solve, this we first need to "unfold" the filters of the formula, which is done by rewriting each subformula recursively in the following way:

- $\varphi' \text{ FILTER } \alpha_1 \land \alpha_2$ is replaced by $(\varphi' \text{ FILTER } \alpha_1) \text{ FILTER } \alpha_2$. 


• $\varphi' \text{FILTER } \alpha_1 \lor \alpha_2$ is replaced by $\varphi' \text{FILTER } \alpha_1 \lor \varphi' \text{FILTER } \alpha_2$.

Notice that, after doing this, all filters have only one predicate, so $p(x)$ can no longer appear inside a propositional disjunction. Now moving down each predicate is done straightforward. For every subformula of the form $\varphi' \text{FILTER } p(x)$, the $p(x)$ filter is removed from $\varphi'$ and instead applied over every subformula of $\varphi'$ with the form $R \text{AS } x$, rewriting it as $R \text{AS } x \text{ FILTER } p(x)$. Because this step moved every predicate to its definition, the resulting formula is clearly in LP normal form, completing the construction.

Now we prove that the construction above satisfies the lemma, i.e., $[[\varphi_{tp}(S)](S) = [[\varphi](S)]$ for every stream $S$, where $\varphi_{tp}$ is the resulting formula after the construction. To prove that the first part does not change the semantics, we show that it stays the same after each iteration. Consider a subformula $\varphi' \text{FILTER } \alpha$ and a predicate $p(x) \in$ unbound($\varphi'$). The only part that is modified is $\varphi_x$, so it suffices to prove that $M \models [[\varphi_x](S,i,\nu)$ holds iff $M \models [[\varphi'_x](S,i,\nu)$. Let $S, i, M, \nu$ be any stream, position, match and valuation, respectively, such that $M \models [[\varphi_x](S,i,\nu)$. If $\nu_x \models p(x)$, then it is enough to prove that $M \models [[\varphi'_x](S,i,\nu)$. In a similar way, the only part in which $\varphi'_x$ differs with $\varphi_x$ is that $p(x)$ was set true in $\alpha$ (let $\alpha'$ be the result of doing this). Therefore, it is enough to prove that, for any $j$, $M'$ and $\varphi'$, if $\nu'_x \equiv p(x)$ holds, then $M' \models [[\varphi'_x](S,i,\nu')$. We know that, when evaluating every predicate to its definition, 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, $\nu'$ has to be equal to $\nu$. The proof for the case $\nu_x \equiv \neg p(x)$ is similar considering $\varphi''_x$ instead of $\varphi'_x$, thus $M \models [[\varphi''_x](S,i,\nu)$ holds.

Now, for the second part we first prove that the “unfolding” does not change semantics, which we do by proving it for each iteration. Consider a stream $S$, a match $M$, an $i \in \mathbb{N}$, a CEPL formula $\rho$, two formulas $\alpha_1, \alpha_2$ and a valuation $\nu$. We prove that $M \models [[\rho \text{FILTER } \alpha_1 \land \alpha_2](S,i,\nu)$ if, and only if, $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$. This is straightforward: $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$ holds if $M \models [[\rho](S,i,\nu)$ and $\nu_x \equiv (\alpha_1 \land \alpha_2)$, which means the same as $M \models [[\rho](S,i,\nu)$ and $\nu_x \equiv (\alpha_1 \land \alpha_2)$, which is the condition for $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$ to hold. Similarly, we prove that $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$ if, and only if, $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$ by definition. $M \models [[\rho \text{FILTER } \alpha_1 \text{FILTER } \alpha_2](S,i,\nu)$ holds if either $M \models [[\rho](S,i,\nu)$ and $\nu_x \equiv (\alpha_1 \land \alpha_2)$, which is the same as $M \models [[\rho](S,i,\nu)$ and $\nu_x \equiv (\alpha_1 \land \alpha_2)$.

Finally, 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 the form $\varphi' \text{FILTER } p(x)$. The same way as before, we focus on the modified part, i.e., we need to prove that $M \models [[\varphi' \text{FILTER } p(x)](S,i,\nu)$ iff $M \models [[\varphi'_p](S,i,\nu)$, where $\varphi'_p$ is the result of adding the filter $p(x)$ for each definition of $x$ inside $\varphi'$, i.e., replace $R \text{AS } x$ with $R \text{AS } x \text{ FILTER } p(x)$ where $R$ is any relation. First, let $S, i, M, \nu$ be any stream, position, match and valuation, respectively, such that $M \models [[\varphi' \text{FILTER } p(x)](S,i,\nu)$, which means that $\nu_x \equiv p(x)$. We know that, when evaluating every subformula $R \text{AS } x$ of $\varphi'$, the valuation $\nu$ must stay the same, because $x \in \text{bound}(\varphi')$. Thus its definition cannot be inside a $+$ operator (notice that if it appears inside a $+$ operator, it represents a value different to $x$, thus the $+$ subformula can be rewritten using a new variable $x'$). Similarly to the reasoning above, it holds that, for any $j$, $M'$ and $\varphi'$, if $\nu'_x \equiv p(x)$, then $M' \models [[R \text{AS } x \text{ FILTER } p(x)](S,j,\nu')$ iff $M' \models [[R \text{AS } x](S,j,\nu')$. Then, because every subformula $R \text{AS } x$ behaves the same, $M \models [[\varphi'_p](S,i,\nu)$ holds. For the opposite direction, let $S, i, M, \nu$ be any stream, position, match and valuation, respectively, such that $M \models [[\varphi'_p](S,i,\nu)$. We prove that $\nu_x \equiv p(x)$ must hold, thus proving that $M \models [[\varphi' \text{FILTER } p(x)](S,i,\nu)$ holds by the same argument as above. By contradiction, assume that $\nu_x \equiv \neg p(x)$. Because we showed that when evaluating every $R \text{AS } x \text{ FILTER } p(x)$ in $\varphi'_p$, the valuation $\nu$ must be the same, the only possible way for $M \models [[\varphi'_p](S,i,\nu)$ to hold is if all $R \text{AS } x$ appear at one side of an OR operator. However, this would contradict the fact that $x \in \text{bound}(\varphi')$, thus $\nu_x \equiv p(x)$ must hold, and so must $M \models [[\varphi' \text{FILTER } p(x)](S,i,\nu)$. Then, $\varphi' \text{FILTER } p(x)$ and $\varphi'_p$ are equivalent, therefore, if we name $\varphi_{tp}$ the result of moving all predicates to their definitions, $[[\varphi_{tp}(S)](S) = [[\varphi](S)]$ for every $S$.

Finally, it is easy to check that the size of $\varphi_{tp}$ will be at most exponential in the size of $\varphi$. Indeed, in each rewriting step (i.e. from $\varphi$ to $\varphi_1$ and from $\varphi_1$ to $\varphi_2$) we can duplicate the size $\varphi$ in the worst case. Since the number of rewriting steps are at most linear in the size of $\varphi$ (if we do the rewriting steps bottom up in the parse tree), we have that $|\varphi_{tp} \in O(2^{|\varphi} \cdot |\varphi|)$. □
C. PROOFS OF SECTION 7

C.1 Proof of Proposition 1

For the following proof consider any two MA $A_1 = (Q_1, \Delta_1, I_1, F_1)$, $A_2 = (Q_2, \Delta_2, I_2, F_2)$ and assume, without loss of generality, that they have disjoint sets of states, i.e., $Q_1 \cap Q_2 = \emptyset$. We first begin by proving closure under union, which is exactly the same as the proof for FSA closure under union. We define the MA $A_1 \cup A_2 = (Q, \Delta, I, F)$ as follows. The set of states is $Q = Q_1 \cup Q_2$, the transition relation is $\Delta = \Delta_1 \cup \Delta_2$; the set of initial states is $I = I_1 \cup I_2$ and the set of final states is $F = F_1 \cup F_2$.

Next we prove closure under intersection. We define the MA $A_1 \cap A_2 = (Q, \Delta, I, F)$ as follows. The set of states is the Cartesian product $Q = Q_1 \times Q_2$; the transition relation is $\Delta = \{(p_1, q_1) \mid (p_1, q_1) \in \Delta_1 \times \Delta_2\}$, that is, both conditions $\alpha_1$ and $\alpha_2$ must be satisfied by the incoming tuple in order to simulate both transitions with the same mark $m$ from $p_1$ to $q_1$ and from $p_2$ to $q_2$ of $A_1$ and $A_2$, respectively; the set of initial states is $I = I_1 \times I_2$ and the set of final states is $F = F_1 \times F_2$.

Now we prove closure under determinization. Define the MA $A_d = (Q_d, \Delta_d, I_d, F_d)$ 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:

$$\Delta_d = \{(T, (\alpha, m), U) \mid \alpha \in F\text{-types}, \text{ and } q \in U \text{ if there is a } p \in T \text{ and } \alpha' \in F_u(R) \text{ such that } (p, (\alpha', m), q) \in \Delta \text{ and } \alpha = \alpha'\}.$$

Here, $F$ is the set of all formulas in the transitions of $\Delta$ and we use the notion of $F$-types defined in the proof of Theorem 6 (see Section D.3.2 for the definition). Finally, the sets of initial and final states are $I_d = \{I\}$ and $F_d = \{T \mid T \in Q_d \wedge T \cap F = \emptyset\}$. The key notion here is the one of $F$-types, which partitions the set of tuples in a way that if a tuple $t$ satisfies a formula $\alpha \in F\text{-types}$, then $\alpha$ implies the conditions of all transition that a run of $A$ could take when reading $t$. This allows us to then apply a determinization algorithm similar to the one for FSA. Notice that $\alpha_1 \equiv \neg \alpha_2$ for every two different formulas $\alpha_1, \alpha_2 \in F\text{-types}$, so the resulting MA $A_d$ is deterministic.

Finally, we prove closure under complementation. Basically, the complementation of a MA is no more than determining it and complementing the set of final states. Formally, we define the MA $A_d^c = (Q, \Delta, I, F)$ as follows. Consider the deterministic MA $\det(A_d) = (Q_d, \Delta_d, I_d, F_d)$. Then, the set of states, the transition relation and the set of initial states are the same as of $\det(A_d)$, i.e., $Q = Q_d$, $\Delta = \Delta_d$ and $I = I_d$, and the set of final states is $F = Q \setminus F_d$.

D. PROOFS OF SECTION 8

D.1 Proof of Theorem 4

For the sake of simplicity, for this proof we will add to the model of MA the ability to have $\epsilon$-transitions. Formally, now a transition relation has the structure $\Delta \subseteq Q \times ((F_u(R) \times \{\bullet, \circ\}) \cup \{\epsilon\}) \times Q$. This basically means the automaton can have transitions of the form $(p, \epsilon, q)$ 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 MA, since any $\epsilon$-transition $(p, \epsilon, q)$ 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 result of Theorem 8 shows that we can rewrite every core CEPL formula as a formula in LP-normal form, so we consider that, if $\varphi$ is not in LP-normal form, then it is first translated into one with an exponential growth from the beginning. We now give a construction that, for every core CEPL formula $\varphi$ in LP-normal form, defines a MA $A$ such that for every match $M$, $M \in [A](S_i)$ iff there exists a valuation $\nu$ such that $M \in [\varphi](S, i, \nu)$ (recall that $S_i$ is the stream $t_i, t_{i+1}, \ldots$). Moreover, we show two properties: (1) for every accepting run $\rho$ there exists a valuation $\nu$ such that every $x \in \text{dom}(\nu)$ appears exactly once in $\rho$ and only at the transition $\nu(x)$ of $\rho$; (2) for every $\nu$ there exists an accepting run $\rho$ of $A$ over $S_i$ such that every $x \in \text{vdef}_x(\varphi)$ that appears in a transition of $\rho$ appears while reading $S[\nu(x)]$. This construction is done recursively in a bottom-up fashion such that, for every subformula, an equivalent MA is built from the MA of its subformulas. Let $\psi$ be any subformula of $\varphi$. Then, the MA $A$ is defined as follows:

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

```
TRUE | \circ
\downarrow
q^i  \beta(x) | \bullet
\downarrow
q^j
```

If $\psi$ has no FILTER the automaton is the same but with $\beta(x) = (\text{type}(x) = R)$.
Now, consider the case times. Then, $\nu$ induction hypothesis, there exists some valuation $j$ accepting run. Notice that match $\rho$ In particular, every $x$ there exist accepting runs such that construction, no direction, consider a match $M = \rho((q', \epsilon, q'\epsilon, \rho(q', \epsilon, q'))$. 

If $\psi = \psi_1 \lor \psi_2$ and $A_1 = (Q_1, \Delta_1, \{q'_1\}, \{q'_1\})$ and $A_2 = (Q_2, \Delta_2, \{q'_2\}, \{q'_2\})$ are the automata for $\psi_1$ and $\psi_2$, respectively, then $A = (Q, \Delta, \{q'\}, \{q'\})$ where $Q$ is the union of the states of $A_1$ and $A_2$ plus the new initial and final states $q', q''$, and $\Delta$ is the union of $\Delta_1$ and $\Delta_2$ plus the empty transitions from $q'$ to the initial states of $A_1$ and $A_2$, and from the final states of $A_1$ and $A_2$ to $q''$. Formally, $Q = Q_1 \cup Q_2 \cup \{q', q''\}$ and $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q', \epsilon, q'), (q', \epsilon, q'\epsilon), (q', \epsilon, q'), (q', \epsilon, q')\}$. 

If $\psi = \psi_1 \land \psi_2$ and $A_1 = (Q_1, \Delta_1, \{q'_1\}, \{q'_1\})$ and $A_2 = (Q_2, \Delta_2, \{q'_2\}, \{q'_2\})$ are the automata for $\psi_1$ and $\psi_2$, respectively, we first define $X = X_1 \cap X_2$, where $X_i = \text{def}((\rho_i))$, and define $X = 2^X$. Now we define the components of $A = (Q, \Delta, \{q'_1\}, \{q'_2\})$. The set of states is $Q = (Q_1 \cup Q_2) \times X$. Then, the transition relation consists of three parts. The first part is $\Delta'_1 = \{(p, Y, (a, m), (q, Y')) | (p, (a, m), q) \in \Delta_1 \text{ and } Y' = Y \cup \{(\alpha, a) \times X)\}$, which allows $A$ to simulate $A_1$, gathering the variables of interest at each transition. The second part is $\Delta'_2 = \{(p, (a, m), (q, Y)) | (p, (a, m), q) \in \Delta_2 \text{ and } Y \cup \{(\alpha, a) \times X)\}$, which allows $A$ to simulate $A_2$ restricting that no variable of interest can be seen again. The third part is $\Delta'_3 = \{(q'_i, Y), (\epsilon, (q'_i, Y)) | Y \in X\}$, which ends the simulation of $A_1$ and begins the one of $A_2$. Then, the transition relation $\Delta$ is defined as $\Delta = \Delta'_1 \cup \Delta'_2 \cup \Delta'_3$. 

If $\psi = \psi_1 + \psi_1$ and $A_1 = (Q_1, \Delta_1, \{q'_1\}, \{q'_1\})$ is the automaton for $\psi_1$, then $A = (Q, \Delta, \{q'_1\}, \{q'_1\})$ where $Q = Q_1$ and $\Delta = \Delta_1 \cup \{(q'_i, \epsilon, q'_i)\}$. Basically, is the same automaton for $\psi_1$ with an $\epsilon$-transition from the final to the initial state.
\( \rho_2 \) is an accepting run of \( \mathcal{A} \), thus if \( M_2 = \text{match}(\rho_2) \) then by induction hypothesis \( M_2 \in [\psi](S, j, \nu) \) for some \( \nu \), where \( j = \max(M_1) \). If \( M = M_1 \cdot M_2 \) then \( M \in \{ \psi : \psi_{+} \} \) \( \{S, i, \nu \} \), thus \( M \in [\psi](S, i, \nu) \). Notice that a valuation \( \nu' \) such that \( \text{dom}(\nu') = \emptyset \) also satisfies \( M \in [\psi](S, i, \nu') \), thus it trivially satisfies property (1). For the other direction, consider a match \( M \) such that \( M \in \{ \psi : \psi_{+} \} \) \( \{S, i, \nu \} \) for some valuation \( \nu \). Then there exists \( \nu' \) such that either \( M \in \{ \psi : \psi_{+} \} \) \( \{S, i, \nu' \} \) or \( M \in \{ \psi_{+} \} \) \( \{S, i, \nu' \} \) \( \{U \} \). We now prove, by induction over the number of iterations, that \( M \in [\psi](S, i, \nu) \) \( \{S, i, \nu' \} \) and, by induction hypothesis, \( M \in \{ \psi \} \{S, i, \nu \} \). If there is just one iteration, then \( M \in \{ \psi \} \{S, i, \nu \} \) \( \{U \} \) and, by induction hypothesis, \( M \in [\psi](S, i, \nu) \) \( \{S, i, \nu' \} \) and \( M \in \{ \psi \} \{S, i, \nu \} \). Notice that \( \text{vdef}_{\psi}(\nu) = \emptyset \) thus every run satisfies property (2).

Finally, it is clear that the size of \( \mathcal{A} \) is linear with respect to the size of \( \varphi \) if \( \varphi \) is safe and already in LP-normal form. However, if \( \varphi \) is not safe, then the construction for \( \varphi \) have an exponential blow-up. Furthermore, if \( \varphi \) is not in LP-normal form, then it is first translated into an equivalent CEPL formula \( \varphi' \) that is adding an exponential growth. Then, \( \mathcal{A}_\varphi \) is of size at most double exponential in \( |\varphi| \) and of size at most linear if \( \varphi \) is safe and in LP-normal form.

\[ \square \]

### D.2 Proof of Theorem 5

For each \( \text{OP} \in \{ \text{AND}, \text{ALL}, \text{UNLESS} \} \) we give a construction for a MA \( \mathcal{A} = (Q, \Delta, I, F) \) that is equivalent to the CEPL formula \( \varphi \) of the form \( \varphi_1 \text{ OP } \varphi_2 \). Moreover, we prove that properties (1) and (2) stated in Theorem 4 still hold. Let \( \mathcal{A}_1 = (Q_1, \Delta_1, I_1, F_1) \) and \( \mathcal{A}_2 = (Q_2, \Delta_2, I_2, F_2) \) be the MA equivalent to \( \varphi_1 \) and \( \varphi_2 \) respectively, define \( X = X_1 \times X_2 \), where \( X = \text{vdef}_{\varphi_1}(\varphi_2) \), and define \( X' = 2^X \).

For the first case the \( \text{OP} \in \{ \text{AND} \} \), the automaton \( \mathcal{A} \) is defined as follows. The set of states is \( Q = Q_1 \times Q_2 \times X \).

Then the set transition relation consists of all transitions of the form \( (p_1, p_2, q_1, \alpha \land \alpha_2, m, q_1, q_2, Y') \) such that there are transitions \( (p_1, q_1, m_1, q_1) \in \Delta_1 \) and \( (p_2, q_2, m_2, q_2) \in \Delta_2 \) such that \( \var{\alpha_1} = \var{\alpha_2} \) \( \var{\alpha} \) \( \var{\alpha_2} \) and \( \var{\alpha} \) \( \var{\alpha_2} \).

Similarly for the second MA, we add transitions \( (p_1, p_2, q_1, \alpha \lor \alpha_2, m, q_1, q_2, Y') \) such that \( (p_1, q_1, m_1, q_1) \in \Delta_1 \) and \( (p_2, q_2, m_2, q_2) \in \Delta_2 \) such that \( \var{\alpha_1} = \var{\alpha_2} \) \( \var{\alpha} \) \( \var{\alpha_2} \) and \( \var{\alpha} \) \( \var{\alpha_2} \).

Now, we define the automaton \( \mathcal{A} \) for the case \( \text{OP} = \text{ALL} \). Basically, the automaton simulates both \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \), allowing them to mark it at the same position. Moreover, if there is a variable in both \( \varphi_1 \) and \( \varphi_2 \), it force them to mark it at the same position.

Before we define the automaton \( \mathcal{A} \) for the case \( \text{OP} = \text{UNLESS} \), we need to define some automaton transformations. Let \( \mathcal{A}^2 = (Q^2, \Delta^2, I^2, F^2) \) be the result of applying to \( \mathcal{A}_2 \) the determinization construction in Proposition 4. Let \( \mathcal{A}^\prime_2 = (Q^2, \Delta^\prime_2, I^2, F^2) \) be a boolean automaton for \( \mathcal{A}_2 \) which is essentially an automaton that gets to an accepting state if there has been a match in \( \mathcal{A}_2 \) in the prefix read until that point. Its structure is the same, except that it has an extra state \( q_a \) which is the only accepting one, i.e., \( Q^\prime_2 = Q_2 \cup \{q_a\}, I^\prime_2 = I_2 \) and \( F^\prime_2 = \{q_a\} \). Because \( \mathcal{A}_2 \) does not return a match, all transitions of \( \Delta_2 \) are copied without their matching symbols, i.e., for every transition \( (p, (\alpha, m), q) \in \Delta_2 \), the transition \( (p, \alpha, q) \) is added in \( \Delta^\prime_2 \). In addition, for every transition in \( \Delta_2 \) of the form \( (p, (\alpha, q), q) \) with \( q \in F_2 \), it adds the transition \( (p, (\alpha, q) \cup \{q_a\}) \) and \( (q_a) = \{q_1^a, q_2^a, \ldots, q_k^a\} \) respectively, where \( q_a \) are associated to the subset \( \{q_1, q_2, \ldots, q_a\} \subseteq Q_2 \). Now we are ready to define \( \mathcal{A} \). The set of states is defined as \( Q = Q_1 \times Q^2_2 \cup (Q_1 \times Q^2_2) \). As notation, we write the states of \( Q^2_2 \) and \( Q^2_2 \) as \( \{q_1^2, q_2^2, \ldots, q_k^2\} \).
q to illustrate that it is associated to a subset of states (of $Q_2$). We define the transition relation as follows. First, for every $(p_1, (α_1, \circ), q_1) \in Δ_1$ and $(p_2, (α_2, \circ), q_2) \in Δ_2$ we add $((p_1, p_2), (α_1 \land α_2, \circ), (q_1, q_2))$ into $Δ$. Second, for every $(p_1, (α_1, \circ), q_1) \in Δ_1$ and $(p_2, α_2, q_2) \in Δ_2$ such that $q_2 \notin F_2$, we add $((p_1, p_2), (α_1 \land α_2, \circ), (q_1, q_2))$ into $Δ$. Finally, for every $(p_1, (α_1, \bullet), q_1) \in Δ_1$, $(p_2, α_2, q_2) \in Δ_2$ we add $((p_1, p_2), (α_1 \land α_2, \bullet), (q_1, q_2))$ into $Δ$, where $p_2$ is the analogous of $p_2$ in $A_2$, i.e., if $p_2 = \{p_2^1, p_2^2, ... , p_2^k\}$, then $p_2' = \{p_2^1, p_2^2, ... , p_2^k\}$. The set of initial states is defined as $I = I_1 \times I_2$ and the set of final states as $F = F_1 \times F_2$. The idea behind this construction is that at the beginning the automaton simulates $A_1$ and $A_2$ with only $\circ$-transitions until $A_1$ marks a position. At this point it goes on with the simulation of $A_1$ and simultaneously verifies that the simulation of $A_2$ can never pass through an accepting state, for which it uses the boolean automaton $A_2$. Notice that we verify this in the construction when we consider only the transitions of $Δ'$ that do not end in an accepting state.

Now, we prove the correctness of the above constructions. First, consider the AND case. Consider a match $M \in [A](S_i)$. Then, there is a run $ρ$ of $A$ of the form:

$$ρ: (q_0, q_0, \emptyset, α_{1/m_1}) \rightarrow (q_1, q_1^2, Y_1) \rightarrow (q_2, q_2, Y_2, α_{2/m_2}) \rightarrow (q_n, q_n, Y_n, α_{n/m_n})$$

Because of the construction, each transition has the form $α = α_2 \land α_2$ such that the runs:

$$ρ_1: q_0 \rightarrow q_1^i \rightarrow q_1^j \rightarrow q_1^k$$

are accepting runs of $A_1$ and $A_2$, respectively, and $M = \text{match}(ρ_1) = \text{match}(ρ_2)$. By induction hypothesis there exist valuations $ν_1$ and $ν_2$ such that $M \in [\varphi_1](S_i, ν_1)$ and $M \in [\varphi_2](S_i, ν_2)$. Moreover, because of property (1), we know that every variable $x \in \text{dom}(ν_1)$ appears exactly once in $ρ_i$ and only at the transition $ν_i(x)$ of $ρ_i$. Because of the construction, no variable $x \in X$ can appear in two different transitions of $ρ$, which means that if it appears at some position of $ρ_1$, then it cannot appear at a different position of $ρ_2$, and conversely. Therefore, for every $x \in \text{dom}(ν_1) \cap \text{dom}(ν_2)$ it holds that $ν_1(x) = ν_2(x)$. If we define $ν = ν_1[ν_2 \rightarrow \text{dom}(ν_2)]$, then $M \in [\varphi_1](S_i, ν)$ and $M \in [\varphi_2](S_i, ν)$ still hold, thus $M \in [\varphi](S_i, ν)$. Moreover, by induction of property (1) over $ρ_1$ and $ρ_2$, and because of the construction, property (1) still holds for $ρ$. For the opposite direction, consider $M \in [\varphi](S_i, ν)$ for some $ν$. By definition it means that $M \in [\varphi_1](S_i, ν)$ and $M \in [\varphi_2](S_i, ν)$ and, by induction hypothesis, there exist accepting runs:

$$ρ_1: q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_n$$

Over $A_1$ and $A_2$, respectively, such that $\text{match}(ρ_1) = \text{match}(ρ_2) = M$. Moreover, because of property (2), we know that every $x \in \text{vdef}(\varphi)$ that appears in a transition of $ρ_1$ appears while reading $S[ν(x)]$. Now, if we define the run:

$$ρ: (q_0, q_0, \emptyset, α_{1/m_1}) \rightarrow (q_1, q_1^2, Y_1) \rightarrow (q_2, q_2, Y_2, α_{2/m_2}) \rightarrow (q_n, q_n, Y_n, α_{n/m_n})$$

Of $A$, where each $α_i = α_1 \land α_2$ and $Y_1 = (α_1) \cap X$, then clearly it is a valid run of $A$, because for every two different transitions, say with $α_1$ and $α_2$, it holds that $α_1 \cap α_2 \neq \emptyset$. Because $\text{match}(ρ) = M$, then $M \in [A](S_i)$. Moreover, by induction of property (2) over $ν$ with $\varphi_1$ and $\varphi_2$, and because of the construction, property (2) still holds for $ρ$.

Consider now the ALL case. Consider a match $M \in [A](S_i)$. Then, there is a run $ρ$ of $A$ of the form:

$$ρ: (q_0, q_0, \emptyset, α_{1/m_1}) \rightarrow (q_1, q_1^2, Y_1) \rightarrow (q_2, q_2, Y_2, α_{2/m_2}) \rightarrow (q_n, q_n, Y_n, α_{n/m_n})$$

Where at some position $j$ and some $i$, for all $k \geq j$ it is the case that $q_k = 1$. Because of the construction, each transition has the form $α_i = α_1 \land α_2$ such that the runs:

$$ρ_1: q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_n$$

Are accepting runs of $A_1$ and $A_2$, respectively, and $M = M_1 \cup M_2$, where $M_1 = \text{match}(ρ_1)$ and $M_2 = \text{match}(ρ_2)$, i.e., $M_1 = \bullet$ only if $m_1 = \bullet$ or $m_2 = \bullet$. Notice that they do not have to end at the same time. By induction hypothesis, there exist valuations $ν_1$ and $ν_2$ such that $M \in [\varphi_1](S_i, ν_1)$ and $M \in [\varphi_2](S_i, ν_2)$. By the same reasoning of the AND case, for every $x \in \text{dom}(ν_1) \cap \text{dom}(ν_2)$ it holds that $ν_1(x) = ν_2(x)$. If we define $ν = ν_1[ν_2 \rightarrow \text{dom}(ν_2)]$, then $M_1 \in [\varphi_1](S_i, ν)$ and $M_2 \in [\varphi_2](S_i, ν)$ still hold, thus $M \in [\varphi](S_i, ν)$. Moreover, by induction of property (1) over $ρ_1$ and $ρ_2$, and because of the construction, property (1) still holds for $ρ$. For the opposite direction, consider $M \in [\varphi](S_i, ν)$ for some $ν$. By definition it means that $M_1 \in [\varphi_1](S_i, ν)$ and $M_2 \in [\varphi_2](S_i, ν)$ for some $M_1$ and $M_2$. Notice that $M_1$ and...
$M_2$ such that $M = M_1 \cup M_2$ and, by induction hypothesis, there exist accepting runs:

$$
\begin{align*}
\rho_1 : q_0 \xrightarrow{a_1/m_1} q_1 \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_n/m_n} q_{n_1} \\
\rho_2 : q_0 \xrightarrow{a_2/m_2} q_1 \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_2/m_2} q_{n_2}
\end{align*}
$$

Over $A_1$ and $A_2$, respectively, such that match($\rho_1$) = $M_1$ and match($\rho_2$) = $M_2$. Moreover, because of property (2) and by the same reasoning of the AND case, then the run:

$$\rho : (q_0, q_0, \rho) \xrightarrow{a_1/m_1} (q_1, q_1, Y_1) \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_n/m_n} (q_{n_1}, q_{n_2}, Y_n)$$

Where $m_j = \bullet$ only if $m_1^j = \bullet$ or $m_2^j = \bullet$, is a valid run of $A$. Because match($\rho$) = $M_1 \cup M_2 = M$, then $M \in [A](S_i)$. Moreover, by induction of property (2) over $\nu$ with $\varphi_1$ and $\varphi_2$, and because of the construction, property (2) still holds for $\rho$.

Consider now the UNLESS case. Consider a match $M \in [A](S_i)$. Then, there is an accepting run $\rho$ of $A$ of the form:

$$\rho : (q_0, q_0) \xrightarrow{a_1/m_1} (q_1, q_1) \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_n/m_n} (q_n, q_n)$$

Then, $\rho$ can be split at position $j = \min(M)$ as $\rho = \rho_1 \cdot \rho_2$ such that $\rho_1$ simulates runs of $A_1$ and $A_2$ simultaneously with only $\circ$ transitions and $\rho_2$ simulates runs of $A_1$ and $A'_2$ simultaneously. Moreover, because of the construction, the run:

$$\rho_1 : q_0 \xrightarrow{a_1/m_1} q_1 \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_n/m_n} q_n$$

Is an accepting run of $A_1$ over $S_i$. Therefore, match($\rho_1$) = $M \in [A_1](S_i)$, thus $M \in [\varphi_1](S, i, \nu)$ for some $\nu$. Now, by contradiction consider that there exist some $M'$ and $\nu'$ such that $M' \in [\varphi_2](S_i)$, with $\min(M) \leq \min(M')$ and $\max(M') \leq \max(M)$. Then, there exist the accepting runs:

$$\sigma^d : q_0 \xrightarrow{\beta_1^d/m_1} q_1 \xrightarrow{\beta_2^d/m_2} \cdots \xrightarrow{\beta_k^d/m_k} q_k$$

$$\sigma^b : q_0 \xrightarrow{\beta_1^b/m_1} q_1 \xrightarrow{\beta_2^b/m_2} \cdots \xrightarrow{\beta_k^b/m_k} q_k$$

Of $A_d$ and $A_b$, respectively such that $k \leq n$, $m_i^j = \circ$ for all $i < j$ (recall that they are the only possible runs for that prefix of $S$ because they are both deterministic). Then, because $\min(M) \leq \min(M')$ and $\max(M') \leq \max(M)$, for every run of $A$ of the form:

$$\rho' : (q_0', q_0) \xrightarrow{a_1/m_1} (q_1', q_1) \xrightarrow{a_2/m_2} \cdots \xrightarrow{a_k/m_k} (q_k', q_k)$$

Such that $m_i^j = \circ$ for all $i < j$, it holds that $q_k' \in F_b$. However, because of the construction there is no transition in $\Delta$ that gets to a state of the form $(p, q)$ with $q \in F_b$, which is a contradiction. Therefore, there is no $M'$ and $\nu'$ such that $M' \in [\varphi_2](S_i)$, with $\min(M) \leq \min(M')$ and $\max(M') \leq \max(M)$, thus $M \notin [\varphi](S, i, \nu)$. The proof for the converse case follows directly from this one, it consists of following the steps in the opposite direction. Clearly, properties (1) and (2) still hold by induction hypothesis, only by keeping the same $\nu$ and $\rho$ of the induction, respectively.

### D.3 Proof of Theorem 6

#### D.3.1 STRICT operator

Consider a MA $A = (Q, \Delta, I, F)$. We will first define a MA $A_{\text{strict}} = (Q_{\text{strict}}, \Delta_{\text{strict}}, I_{\text{strict}}, F_{\text{strict}})$ and then prove that it is equivalent to $\text{strict}(A)$. The set of states is defined as $Q_{\text{strict}} = \{q^m \mid q \in Q \land m \in \{\bullet, \circ\}\}$, the transition relation is $\Delta_{\text{strict}} = \{(p^m, (\alpha, m), q^m) \mid (p, (\alpha, m), q) \in \Delta\} \cup \{(p^\circ, (\alpha, \bullet), q^\circ) \mid (p, (\alpha, \bullet), q) \in \Delta\}$, the initial states are $I_{\text{strict}} = \{q^\circ \mid q \in I\}$ and the final states are $F_{\text{strict}} = \{q^\circ \mid q \in F\}$. Basically, there are two copies of $A$, the first one which only have the $\circ$ transitions, and the second one which only have the $\bullet$ ones, and at any $\circ$ transition it can move from the first on to the second. On the other hand, $A_{\text{strict}}$ starts in the first copy of $A$, 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 $A$, 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, $[A_{\text{strict}}](S) = [\text{strict}(A)](S)$ for every $S$. Let $S$ be any string. First, consider a match $M \in [\text{strict}(A)](S)$. This means that $M \in [A](S)$ and that $M$ has the form $M = \{m_0, m_1, \ldots, m_k\}$ with $m_0 = m_1 + 1$. Therefore, there is an accepting run of $A$ of the form:

$$\rho : q_0 \xrightarrow{a_1/\circ} q_1 \xrightarrow{a_2/\circ} \cdots \xrightarrow{a_m/\circ} q_{m-1} \xrightarrow{a_1/\bullet} q_m \xrightarrow{a_2/\bullet} \cdots \xrightarrow{a_k/\bullet} q_k$$

Such that match($\rho$) = $M$. Consider now the run over $A_{\text{strict}}$ of the form:

$$\rho' : q_0 \xrightarrow{a_1/\circ} q_1 \xrightarrow{a_2/\circ} \cdots \xrightarrow{a_m/\circ} q_{m-1} \xrightarrow{a_1/\bullet} q_m \xrightarrow{a_2/\bullet} \cdots \xrightarrow{a_k/\bullet} q_k$$

It is clear that all transitions of $\rho'$ are in $\Delta_{\text{strict}}$, because the ones with $\circ$ are in the first copy of $A$, 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 $\rho'$ is indeed
run of $A_{\text{strict}}$ over $S$, and because $q_{mk} \in F$, then $q'_{mk} \in F$ and $\rho'$ is an accepting run. Moreover, match($\rho'$) = $M'$, thus $M' \in [A_{\text{strict}}](S)$.

Now, consider a match $M \in [A_{\text{strict}}](S)$, of the form $M = \{m_0, m_1, \ldots, m_k\}$. It means that there is an accepting run of $A_{\text{strict}}$ of the form:

$$\rho: q_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{m-1}} q_{m-1} \xrightarrow{\bullet} q_m \xrightarrow{\alpha_m} \cdots \xrightarrow{\alpha_{m+k}} q_{m+k}$$

Such that match($\rho$) = $M$. Notice that $\rho$ must have this form because of the structure of $A_{\text{strict}}$, which force $\rho$ to have $\circ$ transitions at the beginning and $\bullet$ ones at the end. Consider then the run of $A$ of the form:

$$\rho': q_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{m-1}} q_{m-1} \xrightarrow{\bullet} q_m \xrightarrow{\alpha_m} \cdots \xrightarrow{\alpha_{m+k}} q_{m+k}$$

Similar to the converse case, it is clear that all transitions in $\rho'$ are in $\Delta$. Therefore $\rho'$ is an accepting run of $A$ over $S$, and because match($\rho'$) = $M$, it holds that $M \in [\text{strict}(A)](S)$.

Finally, notice that $A_{\text{strict}}$ consists in duplicating $A$, thus the size of $A_{\text{strict}}$ is two times the size of $A$. $\square$

### D.3.2 NXT operator

Let $R$ be a schema and $A = (Q, \Delta, I, F)$ be a match automaton over $R$. In order to define the new match automaton $A_{\text{EXT}} = (Q_{\text{EXT}}, \Delta_{\text{EXT}}, I_{\text{EXT}}, F_{\text{EXT}})$ we first need to introduce some notation. We begin by imposing an arbitrary linear order $< \ $ between tuples such that, for every two different states $p, q \in Q$, either $p < q$ or $q < p$. Let $T_1 \ldots T_k$ be a sequence of sets of states such that $T_i \subseteq Q$. We say that a sequence $T_1 \ldots T_k$ is a total preorder over $Q$ if $T_i \cap T_j = \emptyset$ 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 $i < j$. For the sake of simplification, we define the concatenation between sets of states such that $T_i \cdot T_j = T_{i,j}, T_i \cap T_j = \emptyset$ whenever $T_i$ and $T_j$ are non-empty and $T_i \cdot T_j = T_i \cup T_j$ otherwise. The concatenation between sets will help to remove empty sets during the final construction. Now, given any sequence $T_1 \ldots T_k$ (not necessarily a total preorder), one can convert $T_1 \ldots T_k$ into a total preorder by applying the operation Total Pre-Ordering (TPO) defined as follows:

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

Let $F = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be the set of all condition formulas in the transitions of $\Delta$. Define the equivalence relation $\equiv_F$ between tuples such that, for every pair of tuples $t_1$ and $t_2$, $t_1 \equiv_F t_2$ holds if, and only if, both satisfy the same formulas, i.e., $t_1 \equiv \alpha_i$ holds iff $t_2 \equiv \alpha_i$ holds, for every $i$. Moreover, for every tuple $t$ let $[t]_F$ represent the equivalence class of $t$ defined by $\equiv_F$, that is, $[t]_F = \{t' \mid t \equiv_F t'\}$. 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 formulas in $F$, i.e., $2^{|F|}$. Now, for every $t$, define the formula:

$$\alpha_t = (\bigwedge_{\alpha_i \in \alpha} \alpha_i) \land (\bigwedge_{t \notin \alpha_i} \neg \alpha_i)$$

and define the new set of formulas $F'=\{\alpha_t \mid t \in \text{tuples}(R)\}$. Notice that for every tuple $t$ there is exactly one formula in $F'$-types that is satisfied by $t$, and that formula is precisely $\alpha_t$. Finally, we extend the transition relation $\Delta$ as a function such that:

$$\Delta(T, \alpha, m) = \{q \in Q \mid \text{exists } p \in T \text{ and } \alpha' \in F_u(R) \text{ such that } \alpha \equiv \alpha' \text{ and } (p, (\alpha', m), q) \in \Delta\}$$

for every $T \subseteq Q$, $\alpha \in F'$-types, and $m \in \{\circ, \bullet\}$.

In the sequel, we define the match automaton $A_{\text{EXT}} = (Q_{\text{EXT}}, \Delta_{\text{EXT}}, I_{\text{EXT}}, F_{\text{EXT}})$ component by component. First, the set of states $Q_{\text{EXT}}$ is defined as follows:

$$Q_{\text{EXT}} = \{(T_1 \ldots T_k, p) \mid T_1 \ldots T_k \text{ is a total preorder over } Q \text{ and } p \in T_i \text{ for some } 1 \leq i \leq k\}$$

Intuitively, the state $p$ is the current state of the ‘simulation’ of $A$ and the sets $T_1 \ldots T_k$ 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 the next-match semantic, e.g., if a run $\rho_1$ reach the state $\{1, 2\} \{3\}$, and other run $\rho_2$ reach the state $\{1, 2\} \{3\}$, then match($\rho_2$) < match($\rho_1$). This property is proven later in Lemma$^8$

Secondly, the transition relation is defined as follows. Consider $\alpha \in F'$-types, $m \in \{\circ, \bullet\}$ and $(T, p, (U, q)) \in Q_{\text{EXT}}$ where $T = T_1 \ldots T_k$ and $p \in T_i$ for some $1 \leq i \leq k$. Then we have that $((T, p), (\alpha, m, (U, q))) \in \Delta_{\text{EXT}}$ if, and only if,

1. $(p, \alpha', m, q) \in \Delta$ for some $\alpha'$ such that $\alpha \equiv \alpha'$,
2. $q \notin \Delta(T_j, \alpha, m')$ for every $m' \in \{\circ, \bullet\}$ and $j < i$,
3. $U = \text{TPO}(U_1^j \cdot U_2^j \cdots U_k^j)$ where $U_j^j = \Delta(T_j, \alpha, \bullet)$ and $U_j^j = \Delta(T_j, \alpha, \circ)$ for $1 \leq j \leq k$, and
4. $q \notin \Delta(T_i, \alpha, \bullet)$ when $m = \circ$,
5. \((p', \alpha', m, q) \notin \Delta\) for every \(p' \in T_i\) such that \(p' < p\) and every \(\alpha'\) such that \(\alpha \neq \alpha'\).

Intuitively, the first condition 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 needed to ensure that there are no two runs \(\rho_1\) and \(\rho_2\) that end in the same state such that \(\text{match}(\rho_1) = \text{match}(\rho_2)\).

Finally, the initial set \(S_{\text{INT}}\) is defined as all states of the form \((I, q)\) where \(q \in I\) and the final set \(F_{\text{INT}}\) 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 \(T_j \cap F = \emptyset\) for all \(j < i\).

Let \(S = t_1 t_2 \ldots\). To prove that the construction is correct, we will need the following lemma.

**Lemma 8.** Consider a MA \(A = (Q, \Delta, I, F)\), a stream \(s\), two states \((T, q), (T', q') \in Q_{\text{INT}}\) with the same sequence \(T = T_1 \ldots T_k\) such that \(p \in T_i, q \in T_j\) for some \(i\) and \(j\), and two runs \(\rho_1, \rho_2\) of \(A_{\text{INT}}\) over \(S\) that have the same length and reach the states \((T, q)\) and \((T, q')\), respectively. Then, \(\rho_1 \neq \rho_2\) and only if:

\[
\text{match}(\rho_2) \leq_{\text{next}} \text{match}(\rho_1)
\]

**Proof.** We will prove by induction over the length of the runs. Let \(q_0, q'_0 \in I\) be any two initial states of \(A\), not necessarily different. First, assume that both runs consist of reading a single tuple \(t\). Then, the runs are of the form:

\[
\rho_1 : (I, q_0) \xrightarrow{\alpha_1/m_1} (T, p) \quad \text{and} \quad \rho_2 : (I, q'_0) \xrightarrow{\alpha_1/m_2} (T, q')
\]

where \(T = T_1 T_2 = \text{TPO}(\Delta(I, \alpha_1, \bullet) \Delta(I, \alpha_1, \circ))\) 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 \(\text{match}(\rho_1) = \{\}\) and \(\text{match}(\rho_2) = \{\},\) 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, \(\text{match}(\rho_1) = \{\}\) and \(\text{match}(\rho_2) = \{\},\) therefore \(\text{match}(\rho_2) \leq_{\text{next}} \text{match}(\rho_1)\). Now, let \(S = t_1 t_2 \ldots t_n \ldots\) and consider that the runs are of the form:

\[
\rho_1 : (I, q_0) \xrightarrow{\alpha_1/m_1} (T_1, q_1) \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_{n-1}/m_{n-1}} (T_{n-1}, q_{n-1}) \xrightarrow{\alpha_n/m_n} (T, p) \\
\rho_2 : (I, q'_0) \xrightarrow{\alpha_1/m'_1} (T_1, q'_1) \xrightarrow{\alpha_2/m'_2} \ldots \xrightarrow{\alpha_{n-1}/m'_{n-1}} (T_{n-1}, q'_{n-1}) \xrightarrow{\alpha_n/m'_n} (T, q')
\]

Notice that both runs have the same sequences \(T_1, \ldots, T_{n-1}\) because each sequence \(T_i\) is defined only by the previous sequence \(T_{i-1}\) and the tuple \(t_i\) which implicitly defines the formula \(\alpha_{t_i}\). Furthermore, all the runs over the same word must have the same sequences. Define the runs \(\rho_1'\) and \(\rho_2'\), respectively, as the runs \(\rho_1\) and \(\rho_2\) without the last transition. Consider that \(T_{n-1}\) has the form \(T_{n-1} = U_1 U_2 \ldots U_k\), and that \(q_{n-1} = U_r\) and \(q'_{n-1} = U_s\) for some \(r\) and \(s\). Notice that, because of the construction, if it is the case that \(r < s\) (\(r > s\) resp.) then \(i < j\) (\(i > j\) resp.) must hold. For the if direction, consider that \(\text{match}(\rho_2') \leq_{\text{next}} \text{match}(\rho_1')\). If \(\text{match}(\rho_1') = \text{match}(\rho_2')\), 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 \(\text{match}(\rho_2') \leq_{\text{next}} \text{match}(\rho_1')\), by induction hypothesis it means that \(r < s\) and because of the construction, \(i < j\). Notice that \(\text{match}(\rho_1') \leq_{\text{next}} \text{match}(\rho_2')\) cannot occur because the lower element of \(\text{match}(\rho_2')\) not in \(\text{match}(\rho_1')\) would still be the lower element of \(\text{match}(\rho_2)\) not in \(\text{match}(\rho_1)\), thus contradicting \(\text{match}(\rho_2) \leq_{\text{next}} \text{match}(\rho_1)\). 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 \(\text{match}(\rho_1) = \text{match}(\rho_1') \cup \{n\}\) and \(\text{match}(\rho_2) = \text{match}(\rho_2')\). By induction hypothesis, \(\text{match}(\rho_1') = \text{match}(\rho_2')\), therefore \(\text{match}(\rho_2) \leq_{\text{next}} \text{match}(\rho_1)\). Consider now the case that \(r < s\). By induction hypothesis, \(\text{match}(\rho_2') \leq_{\text{next}} \text{match}(\rho_1')\) and, because the last transition can only add \(n\) to both matches, it follows that \(\text{match}(\rho_2) \leq_{\text{next}} \text{match}(\rho_1)\). □

Now, we need to prove that if \(M \in [\text{NXT}(A)](S)\), then \(M \in [\text{A}_{\text{INT}}](S)\) and vice versa. First, consider a match \(M \in [\text{A}_{\text{INT}}](S)\). To prove that \(M \in [\text{NXT}(A)](S)\), we need to show that \(M \in [A](S)\) and that for all matches \(M'\) such that \(M \leq_{\text{next}} M'\) and \(M = \max(M')\), \(M' \notin [A](S)\). Assume that the run associated to \(M\) is:

\[
\rho : (U_0, q_0) \xrightarrow{\alpha_1/m_1} (U_1, q_1) \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_n/m_n} (U_n, q_n)
\]

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

\[
\rho' : q_0 \xrightarrow{\alpha_1/m_1} q_1 \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_n/m_n} q_n
\]

is an accepting run of \(A\) over \(S\), and thus \(M \in [A](S)\). Now, recall from construction of \(F_{\text{INT}}\) that there exists \(i \leq k\) such that \(q_0 \in T_i\) and \(T_j \cap F = \emptyset\) for all \(j < i\), where \(T_1 \ldots T_k = U_n\). Then, because of Lemma 8, \(M' \leq_{\text{next}} M\) for every other \(M' \in [A](S)\) such that \(\max(M) = \max(M')\), otherwise the run of \(M'\) would end in a state inside a \(T_j\) such that \(j < i\) which cannot happen. Therefore, \(M \in [\text{NXT}(A)](S)\).
Now, consider a match $M \in [\text{NXT}(A)](S)$. Assume that the run associated to $M$ is:

$$\rho: q_0 \xrightarrow{\alpha_1/m_1} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_n/m_n} q_n$$

To prove that $M \in [\text{A}_{\text{NXT}}](S)$ we will prove that there exists an accepting run on $A_{\text{NXT}}$. Based on $\rho$, consider now the run:

$$\rho': (U_0, p_0) \xrightarrow{\alpha'_1/m'_1} (U_1, p_1) \xrightarrow{\alpha'_2/m'_2} \cdots \xrightarrow{\alpha'_n/m'_n} (U_n, p_n)$$

Where the matches $m_1, \ldots, m_n$ are the same, each condition $\alpha_i$ is defined by $t_i$ and each $U_i$ is the result of applying the function TPO based on $U_{i-1}$ and $\alpha_i$. Moreover, each $p_i$ is defined as follows. As notation, consider that $U_i = T_1 \cdots T_k$ and that every $q_i$ is in the $r_i$-th set of $U_i$, i.e., $q_i \in T_{r_i}$. Then, $p_i$ is the lower state in $T_{r_i}$ such that $(p_i, (\alpha_{i+1}, m_{i+1}, p_{i+1})) \in m_i$. Notice that $\rho'$ is completely defined by $\rho$ and $S$. We will prove that $\rho'$ is an accepting run by checking that all transitions meet the conditions of the transition relation $\Delta_{\text{NXT}}$. Now, it is clear that the first condition is satisfied by all transitions, i.e., for every $i$ it holds that $(p_{i-1}, \alpha', m_i, p_i) \in \Delta$ for some $\alpha'$ such that $\alpha_i \equiv \alpha'$ (just consider $\alpha' = \alpha_i$). For the second condition, by contradiction suppose that it is not satisfied by $\rho'$. It means that for some $i$, $p_i \in \Delta(T_j^{-1}, \alpha, m')$ for some $m' \in \{\bullet, \circ\}$ and $j < r_i$. In particular, consider that the state $p_i \in T_j^{-1}$ is the one for which $(\rho', \alpha, m') \in \Delta$. 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:

$$\sigma: q_0 \xrightarrow{\alpha'_1/m'_1} q_1 \xrightarrow{\alpha'_2/m'_2} \cdots \xrightarrow{\alpha'_n/m'_n} (U_n, p_n)$$

Where $p_i'$ are defined in a similar way to $p_i$. Define for every run $\gamma$ and every $i$ the run $\gamma_i$ as $\gamma$ until the $i$-th transition. For, $p_i$ is equal to the run $\rho$ until the state $q_i$. Then, by Lemma 8, match $\rho' < \text{match}(\sigma')$, but match $\rho' = \text{match}(\sigma')$. This is a contradiction, since $\text{match}(\rho')$ and $\text{match}(\sigma')$ differ from $\text{match}(\rho'_{i-1})$ and $\text{match}(\sigma'_{i-1})$ in that the latter can contain additional positions from $i$ to $n$, but the minimum position remains in $\text{match}(\rho'_{i-1})$, and therefore in $\text{match}(\sigma')$. The fourth condition is proven by contradiction too. Suppose that it is not satisfied by $\rho'$, which means that for some $i$, $p_i \in \Delta(T_i^{-1}, \alpha_{i+1}, \bullet)$ when $\alpha_i = \alpha$. Then, the run:

$$\sigma: q_0 \xrightarrow{\alpha'_1/m'_1} \cdots \xrightarrow{\alpha'_i/m'_i} q_i \xrightarrow{\alpha'_i/m'_i} \cdots \xrightarrow{\alpha'_n/m'_n} (U_n, p_n)$$

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

Now, we analyze the properties of the automaton $A_{\text{NXT}}$. First, we show that $[\text{A}_{\text{NXT}}]$ is at most exponential over $[A]$. Notice that each state in $Q_{\text{NXT}}$ 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 are no more than $2^{2|O|} |Q|$ possible states in $Q_{\text{NXT}}$, thus $|\text{A}_{\text{NXT}}| \in \mathcal{O}(2^{|A|})$.

### D.3.3 MAX operator

Let $A = (Q, \Delta, q_0, F)$ be a match automaton. Without loss of generality, we assume that $A$ is deterministic. If not, one can determinize $A$ incurring in an exponential blow-up in the number of states. Similarly to the construction of MA for the NXT, we define the set $F$-types such that for every tuple $t$ there is exactly one formula $\alpha_t$ in $F$-types that is satisfied by $t$, and extend the transition relation $\Delta$ as a function $\Delta(T, \alpha, m)$ for every $T \subseteq Q$, $\alpha \in F$-types, and $m \in \{\bullet, \circ\}$. Similarly, we overload the notation of $\Delta$ as a function such that $\Delta(T, \alpha) = \Delta(T, \alpha, \bullet) \cup \Delta(T, \alpha, \circ)$. We define the match automaton $A_{\text{MAX}} = (Q_{\text{MAX}}, \Delta_{\text{MAX}}, I_{\text{MAX}}, F_{\text{MAX}})$ such that $Q_{\text{MAX}} = Q \times 2^Q$, $I_{\text{MAX}} = \{(q_0, \emptyset)\}$, and $F_{\text{MAX}} = \{(q, T) \in Q_{\text{MAX}} | q \in F \land T \cap F = \emptyset\}$. For the transition relation $\Delta_{\text{MAX}}$ we distinguish two cases depending on whether the transition is marking or not. For the unmarking transition we have that $((p, T), (\alpha, \circ), (q, U)) \in \Delta_{\text{MAX}}$ iff $U = \Delta(T, \alpha) \cup \Delta(T, \alpha, \bullet)\cup \Delta(T, \alpha, \circ)$, $q \notin U$ and there is a formula $\alpha' \in F_a(R)$ such that $(p, \alpha', q) \in \Delta$ and $\alpha = \alpha'$. For every $(p, T), (q, U) \in Q_{\text{MAX}}$ and $\alpha \in F$-types.

Next, we prove the above, i.e., $M \in [\text{MAX}(A)](S)$ iff $M \in [\text{A}_{\text{MAX}}](S)$. First, we prove the if direction. Consider a match $M$ such that $M \in [\text{A}_{\text{MAX}}](S)$. To prove that $M \in [\text{MAX}(A)](S)$, we first prove that $M \in [A](S)$ by giving an accepting run of $A$ associated to $M$. Assume that the run of $\text{A}_{\text{MAX}}$ over $S$ associated to $M$ is:

$$\rho: (q_0, T_0) \xrightarrow{\alpha_1/m_1} (q_1, T_1) \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_n/m_n} (q_n, T_n)$$

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

$$q_0 \xrightarrow{\alpha_1/m_1} \cdots \xrightarrow{\alpha_n/m_n} q_n$$
Is an accepting run of $A$ associated to $M$. Now, we prove by contradiction that for every $M'$ such that $M \subset M'$, $M' \notin [A](S)$. 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 9.** Consider a deterministic match automaton $A = (Q, \Delta, I, F)$, a stream $S = t_1, t_2, \ldots$, a partial run of $A_{\text{MAX}}$ over $S$:

$$\sigma : (q_0, T_0) \xrightarrow{\alpha_1/m_1} (q_1, T_1) \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_n/m_n} (q_n, T_n)$$

And a partial run of $A$ over $S$:

$$\sigma' : p_0 \xrightarrow{\alpha_1/m_1'} p_1 \xrightarrow{\alpha_2/m_2'} \cdots \xrightarrow{\alpha_n/m_n'} p_n$$

Then, if $p_0 \in T_0$ and $m_i' = \bullet$ at every $i$ for which $m_i = \circ$, 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 $m_n' = \displaystyle\bullet$ too, thus $(p_{n-1}, (\alpha_n, \bullet), p_n) \in \Delta$. Furthermore, $T_n = \Delta(T_{n-1}, \alpha_{n, \bullet})$ and therefore $p_n \in T_n$, because $p_{n-1} \in T_{n-1}$. Now, consider the case $m_n = \circ$. Either $(p_{n-1}, (\alpha_n, \bullet), p_n) \in \Delta$ or $(p_{n-1}, (\alpha_n, \circ), p_n) \in \Delta$, so $p_n \in \Delta(T_{n-1}, \alpha_{n, \circ})$. Moreover, $\Delta(T_{n-1}, \alpha_{n, \circ}) \subseteq T_n$ because of the construction of $A_{\text{MAX}}$, therefore $p_n \in T_n$. \(\square\)

Now, by contradiction consider a match $M'$ such that $M \subset M'$ and $M' \notin [A](S)$. Then, there must exist an accepting run of $A$ over $S$ associated to $M'$ of the form:

$$\rho : p_0 \xrightarrow{\alpha_1/m_1'} p_1 \xrightarrow{\alpha_2/m_2'} \cdots \xrightarrow{\alpha_n/m_n'} p_n$$

such that $m_i' = \bullet$ at every $i$ for which $m_i = \circ$, 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 $A$ is deterministic, $\rho$ can be rewritten as:

$$\rho : q_0 \xrightarrow{\alpha_1/m_1} \cdots \xrightarrow{\alpha_i/m_i} q_i \xrightarrow{\alpha_i'/m_i'} p_i \xrightarrow{\alpha_{i+1}/m_{i+1}} \cdots \xrightarrow{\alpha_n/m_n'} p_n$$

Similarly, to ease visualization we rewrite $\rho$ as:

$$\rho : (q_0, T_0) \xrightarrow{\alpha_1/m_1} \cdots \xrightarrow{\alpha_i/m_i} (q_i, T_i) \xrightarrow{\alpha_i'/m_i'} (q_i', T_{i'}) \xrightarrow{\alpha_{i+1}/m_{i+1}} \cdots \xrightarrow{\alpha_n/m_n} (q_n, T_n)$$

In particular, the transition $((q_{i-1}, T_{i-1}), (\alpha_{i-1}, \circ), (q_i, T_i))$ is in $\Delta_{\text{MAX}}$, which means that $\Delta \subseteq \Delta_{\text{MAX}}$, which means that $\Delta_{\text{MAX}}$ is non-empty. Moreover, $(q_{i-1}, (\alpha_i', \bullet), p_i) \in \Delta \Delta_{\text{MAX}}$. But, because $\rho$ is an accepting run, $T_n \cap F = \emptyset$ and so $p_n \notin F$, which is a contradiction to the statement that $\rho$ is an accepting run. Therefore, for every $M'$ such that $M \subset M'$, $M' \notin [A](S)$, hence $M \notin [\text{MAX}(A)](S)$.

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

**Lemma 10.** Consider a deterministic match automaton $A = (Q, \Delta, I, F)$, a stream $S = t_1, t_2, \ldots$, a run of $A_{\text{MAX}}$ over $S$:

$$\sigma : (q_0, T_0) \xrightarrow{\alpha_1/m_1} (q_1, T_1) \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_n/m_n} (q_n, T_n)$$

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

$$\sigma' : p_0 \xrightarrow{\alpha_1/m_1'} p_1 \xrightarrow{\alpha_2/m_2'} \cdots \xrightarrow{\alpha_n/m_n'} p_n$$

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

**Proof.** It will be proved by induction over the length $n$. The base case is $n = 0$, which is trivially true because $T_0 = \emptyset$. Assume now that the Lemma holds for $n - 1$. Define the run $\sigma_{n-1}$ as the run $\sigma$ without the last transition. For any state $q \in T_{n-1}$, let $\sigma_q'$ be the run that ends in $q$ such that $\text{match}(\sigma_{n-1}) \subset \text{match}(\sigma_q')$. Consider the case $m_n = \circ$. Then, either $p \notin \Delta(T_{n-1}, \alpha_{n, \circ})$ or $p \notin \Delta((q_{n-1}, (\alpha_{n, \circ}, p))$. In the former scenario, there must be a $q \in T_{n-1}$ and $\alpha \in F_n(R)$ such that $(q, (\alpha, m, p)) \notin \Delta$ and $\alpha_{n, \circ} \vdash \alpha$, with $m \in \{\bullet, \circ\}$. Define $\sigma'$ as the run $\sigma_{n-1}'$ followed by the transition $(q, (\alpha, m, p)$. Then $\sigma'$ satisfies $\text{match}(\sigma) \subset \text{match}(\sigma')$. In the latter scenario, there must be an $\alpha \in F_n(R)$ such that $(q_{n-1}, (\alpha_{n, \bullet}, p) \notin \Delta$ and $\alpha_{n, \circ} \vdash \alpha$. Define $\sigma'$ as $\sigma_{n-1}$ followed by the transition $(q_{n-1}, (\alpha_{n, \bullet}, p)$. Then $\sigma'$ satisfies $\text{match}(\sigma) \subset \text{match}(\sigma')$. Finally, the Lemma holds for every $n$. \(\square\)

Consider a match $M$ such that $M \in [\text{MAX}(A)](S)$. This means that there is an accepting run of $A$ over $S$ associated to $M$. Define that run as:

$$\rho : q_0 \xrightarrow{\alpha_1/m_1} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_n/m_n} q_n$$
Where \( q_0 \in I, q_n \in F \) and \( (q_{i-1}, (\alpha_i, m_i), q_i) \in \Delta \). To prove that \( M \in [\mathcal{A}_{\mathsf{MAX}}](S) \) we give an accepting run of \( \mathcal{A}_{\mathsf{MAX}} \) over \( S \) associated to \( M \). Consider the run:

\[
\rho' : (q_0, T_0) \xrightarrow{\alpha_1/m_1} (q_1, T_1) \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_n/m_n} (q_n, T_n)
\]

Where \( T_0 = \emptyset, T_i = \Delta(T_{i-1}, \alpha_i) \cup \Delta((q_{i-1}), \alpha_i, \bullet) \) if \( m_i = \circ \), and \( T_i = \Delta(T_{i-1}, \alpha_i, \bullet) \) if \( m_i = \bullet \). To be a valid run, every transition \((T_{i-1}, \alpha_i, m_i, T_i)\) must be in \( \Delta_{\mathsf{MAX}} \), 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 an \( \alpha_i \) such that \( \alpha_i = \alpha_0 \) and \((q_{i-1}, (\alpha_i, m_i), q_i) \in \Delta \), 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 \( q_i \in \Delta((q_{i-1}), \alpha_i, \bullet) \) or \( q_i \in \Delta(T_{i-1}, \alpha_i) \). In the first scenario, consider a new run \( \sigma \) to be exactly the same as \( \rho \), but changing \( m_i \) with \( \bullet \). Then \( \sigma \) is also an accepting run, and \( \text{match}(\rho) \subseteq \text{match}(\sigma) \), which is a contradiction to the definition of the maximal semantic. In the second scenario, there must be some \( p \in T_{i-1} \) and \( \alpha \in \mathcal{F}_d(R) \) such that \((p, (\alpha, m), q_i) \in \Delta \), where \( m \in \{\bullet, \circ\} \). Because of Lemma 10 it means that there is a run \( \sigma' \) over \( S \):

\[
\sigma' : p_0 \xrightarrow{\alpha_1/m_1} p_1 \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_i/m_i} p_i \xrightarrow{\alpha_i/m_i} \ldots p_{n-2} \xrightarrow{\alpha_{n-1}/m_{n-1}} p
\]

Such that \( \text{match}(p_{i-1}) \subseteq \text{match}(\sigma') \), where \( p_{i-1} \) is the run \( \rho \) until transition \( i-1 \). Moreover, because \((p, (\alpha, m), q_i) \in \Delta \) we can define the run:

\[
\sigma : p_0 \xrightarrow{\alpha_1/m_1} p_1 \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_i/m_i} p \xrightarrow{\alpha/m} q_i \xrightarrow{\alpha_{i+1}/m_{i+1}} \ldots \xrightarrow{\alpha_n/m_n} q_n
\]

Such that \( \text{match}(p_{i-1}) \subseteq \text{match}(\sigma) \), 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 \( q_i \in \Delta(T_{i-1}, \alpha_i, \bullet) \). Then, there must be some \( p \in T_{i-1} \) and \( \alpha \in \mathcal{F}(R) \) such that \((p, (\alpha, \bullet), q_i) \in \Delta \). Alike the previous case, because of Lemma 10 there is a run:

\[
\sigma : p_0 \xrightarrow{\alpha_1/m_1} p_1 \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_i/m_i} p \xrightarrow{\alpha/m} q_i \xrightarrow{\alpha_{i+1}/m_{i+1}} \ldots \xrightarrow{\alpha_n/m_n} q_n
\]

Such that \( \text{match}(p_{i-1}) \subseteq \text{match}(\sigma) \), which is a contradiction. Then, \( q_i \notin T_i \), therefore \((T_{i-1}, (\alpha_i, m_i), T_i) \in \Delta_{\mathsf{MAX}} \) for every \( i \). The above proved that \( \rho' \) is a run of \( \mathcal{A}_{\mathsf{MAX}} \) but is a becoming run it must hold that \( T_0 \cap F = \emptyset \). By contradiction, assume otherwise, i.e., there is some \( q \in Q \) such that \( q \in T_0 \cup F \). Then, because of Lemma 10 there is another accepting run \( \sigma' \) of \( \mathcal{A}_{\mathsf{MAX}} \) over \( S \) such that \( M \subseteq \text{match}(\sigma') \), which contradicts the fact that \( M \) is maximal. Thus, \( T_0 \cap F = \emptyset \) and \( \rho' \) is an accepting run, therefore \( M \in [\mathcal{A}_{\mathsf{MAX}}](S) \).

It is clear that \( \mathcal{A}_{\mathsf{MAX}} \) is of size exponential in the size of \( A \) if this is deterministic, and double exponential if not (because of the exponential cost of determinizing it).

**E. PROOFS OF SECTION 9**

**E.1 Proof of Theorem 7**

To prove the theorem we provide Algorithm 1 and prove that it has the properties of an \( f \)-evaluation strategy. In Algorithm 1 procedure \( \text{Eval} \) evaluates a match automata \( A \) over a stream \( S \), i.e., stores the information of its execution and enumerates the matches after every new event arrives. To keep the algorithm simple, we assumed that \( A \) is deterministic. This is not a necessary condition since one could do the automaton’s determinization on-the-fly by keeping track of the current set of states, similar to traditional FDA, thus avoiding the exponential blow-up of determinization. Other reasonable assumption is that \( A \) is complete, that is, for every state \( q \), event \( e \) and mark \( m \in \{\bullet, \circ\} \) there is a transition \((q, \alpha, m, p)\) in \( A \) such that \( e = \alpha \). This is used at lines 26 and 34 of Algorithm 1. If \( A \) is not complete, one would only have to check first if there exist such transitions before applying \( \text{MoveMarking} \) and \( \text{MoveNotMarking} \).

In Algorithm 1 the basic structure to store the matches’ positions is the \textit{node}. Each node contains four attributes: \textit{time, top, bot, next}. The first one represents the time at which the node was created, and is the data used to compute the matches. The remaining three are pointers to other nodes, which are better explained later. The access points to the data structure are the variables \textit{first}_q and \textit{last}_q. Consider any iteration \( i \). The main idea of the stored data structure is the following: for every state \( q \) there is a list of nodes which contain the matches’ information of all the runs that are in state \( q \) while reading the \( i \)-th event. The first node of the list is \textit{first}_q, and each node points to the following one with it’s attribute \textit{next}. For notation, let \textit{list}_q be such list. For each node \( n \) on the list, the attributes \textit{top} and \textit{bot} are used to access the previous positions of the match, and are used in the following way. Both of them are in the same previous list, and the former appears first, i.e., \textit{bot} can be reached by moving through the list from \textit{top} (using the \textit{next} attribute). Let \textit{prev}_i = n_1, n_2, \ldots, n_k be the nodes between \textit{top} and \textit{bot}, that is, \( n_1 = n.\textit{top}, n_k = n.\textit{bot} \) and \( n_i = n_1.\textit{next} \). Then, each \( n_i.\textit{time} \) contains the position that came before \( n_i.\textit{time} \) for some match. That way, all matches can be computed by recursively moving through all the nodes of \textit{prev}_i. Finally, considering all the nodes of \textit{list}_q, one can compute all matches that end at state \( q \), which is the main idea of the procedure \textit{EnumAll}.

The updating procedure works in the following way. The variables \textit{first}_q and \textit{last}_q are used to store the values of \textit{first}_q and \textit{last}_q for the next iteration, i.e., they define the states’ lists for the next iteration. First, if there is a
Algorithm 1 Evaluate $A = (Q, \delta, q_0, F)$ over a stream $S$

```latex
1: procedure Evaluate($A(S)$) 
2: for all $q \in Q \setminus \{q_0\}$ do 
3: \hspace{1em} last$_q$ ← first$_q$ ← null 
4: end for 
5: last$_{q_0}$ ← first$_{q_0}$ ← $\bot$ 
6: while $c \leftarrow$ yield$_{q_0}$ do 
7: \hspace{1em} for all $q \in Q$ do 
8: \hspace{2em} last$_q'$ ← first$_q'$ ← null 
9: end for 
10: for all $(q, \text{first}_q, \text{null})$ do 
11: \hspace{2em} if $\delta(q, \text{c}, \text{e}) \neq \text{null}$ then 
12: \hspace{3em} MoveMarking($q, \text{c}$) 
13: end if 
14: if $\delta(q, \text{c}, \text{e}) \neq \text{null}$ then 
15: \hspace{2em} MoveNotMarking($q, \text{c}$) 
16: end if 
17: end for 
18: for all $q \in Q$ do 
19: \hspace{1em} first$_q$ ← first$_q'$ 
20: \hspace{1em} last$_q$ ← last$_q'$ 
21: end for 
22: EnumAll(c.time) 
23: end while 
24: end procedure 
25: procedure MoveMarking($q, \text{c}$) 
26: \hspace{1em} $p \leftarrow \delta(q, \text{c}, \text{e})$ 
27: \hspace{1em} $n \leftarrow \text{Node}(c.time, \text{first}_q, \text{last}_q, \text{first}_p')$ 
28: \hspace{1em} first$_p'$ ← $n$ 
29: \hspace{1em} if last$_p' = \text{null}$ then 
30: \hspace{2em} last$_p'$ ← first$_p'$ 
31: end if 
32: end procedure 
33: procedure MoveNotMarking($q, \text{c}$) 
34: \hspace{1em} $p \leftarrow \delta(q, \text{c}, \text{e})$ 
35: \hspace{1em} if last$_p' = \text{null}$ then 
36: \hspace{2em} first$_p' \leftarrow \text{first}_q$ 
37: \hspace{2em} last$_p' \leftarrow \text{last}_q$ 
38: else 
39: \hspace{2em} last$_p'$.next ← first$_q$ 
40: \hspace{2em} last$_p'$ ← last$_q$ 
41: end if 
42: end procedure
```

Algorithm 2 Enumerate matches at time $time$

```latex
1: procedure Enumerate(time) 
2: for all $q \in \mathcal{P} \cap \{q' \mid \text{first}_q' \neq \text{null}\}$ do 
3: \hspace{1em} $n \leftarrow$ first$_q$ 
4: \hspace{1em} while $n \neq \text{null}$ \&\& $n$.time = $time$ do 
5: \hspace{2em} EnumAll($n, \emptyset$) 
6: \hspace{2em} $n \leftarrow n$.next 
7: end while 
8: end for 
9: end procedure 
10: procedure EnumAll(node, match) 
11: if node = $\bot$ then 
12: SendResult(match) 
13: else 
14: \hspace{1em} node$'$ ← node.top 
15: \hspace{1em} while node$'$ = node.bot do 
16: \hspace{2em} EnumAll(node$,\text{match}.add(node$.time$)) 
17: \hspace{2em} node$'$ ← node$.next$ 
18: end while 
19: EnumAll(node$,\text{node}.time + match$) 
20: end if 
21: end procedure
```
transition \((q, \alpha, \bullet, p)\) such that \(e \equiv \alpha\), then a new node \(n\) is added to the list of \(p\). This node contains the position \(e.\text{time}\) and pointers to the first and last nodes of the list of \(q\) in the previous iteration; these values are stored in the attributes \(n.\text{time}\), \(n.\top\) and \(n.\text{bot}\), respectively. Also, if there is a transition \((q, \alpha, \circ, p)\) such that \(e \equiv \alpha\), then all the previous list of \(q\) is added to the list of \(p\). One way to see this is that all the runs that ended at state \(q\) on the previous iteration are now extended with this transition, thus ending now at state \(p\). These two updates are performed by procedures \text{MoveMarking} and \text{MoveNotMarking}.

**Theorem 10.** Consider any \(A\) and \(S\). For every iteration \(i\), Algorithm 1 enumerates \([A]_i(S)\).

**Proof.** To prove this, we first need to verify the following useful properties. As notation, for every \(q\) name \(\text{list}_q\) the nodes from \text{first}_q to \text{last}_q following the next order. Moreover, for every node \(n\), name \(\text{prev}_n\) the list of nodes from \(n.\top\) to \(n.\text{bot}\).

**Proposition 2.** For every two different states \(p\) and \(q\), the nodes of \(\text{list}_p\) and \(\text{list}_q\) are disjoint.

**Proof.** It is not hard to see this from the algorithm. First, clearly they begin disjoint, since some start with value \text{null} and others start with \text{bot}. Then, at each iteration the lists are updated by the \text{MoveMarking} and \text{MoveNotMarking} procedures. The first one only creates new nodes and adds them at the beginning of the list. The second one only moves all the nodes of one list to the end of another one and, because \(A\) is deterministic, these nodes are moved only to one target list (if there were two \circ transitions from the same state \(q\), the list \(\text{list}_q\) would have to be duplicated in the lists of both target states).

**Proposition 3.** For every node \(n\), the list \(\text{prev}_n\) remains the same from one iteration to the next.

**Proof.** The main idea is that the middle of the lists is never modified. The only modifications are: adding new nodes (at the beginning of the list, by procedure \text{MoveMarking}) and merging a series of lists into another list (which are appended at the end of the latter, by procedure \text{MoveNotMarking}). Moreover, the only scenario at which the nodes’ pointers are modified is when lists are appended in the \text{MoveNotMarking}, where the \text{next} pointer of the last node of one list is changed to point to the first node of other list. Since the lists are disjoint, this modification will not alter the middle part of any list. Also, every node created at some iteration has the attributes \text{top} and \text{bot} pointing at the first and last nodes of some list \(\text{list}_n\) (from the previous iteration), respectively. Then, it follows that the list \(\text{prev}_n\) will be equal to the previous \(\text{list}_q\), and since the middle part of the lists is never modified, \(\text{prev}_n\) will not change in the future.

**Proposition 4.** For every node \(n\) and every \(m\), \text{EnumAll}(n, m) returns the same result at every iteration.

**Proof.** This follows straightforward from Proposition 3. The result given by \text{EnumAll}(n, m) is determined only on the result given by \text{EnumAll}(n', n.\text{time} + m) for every \(n'\) on the list \(\text{prev}_n\). Similarly, the result of \text{EnumAll}(n', m'), with \(m' = n.\text{time} + m\), is determined only by the results of \text{EnumAll}(n'', n.\text{time} + m') for every \(n''\) in \(\text{prev}_n'\), and so on. Because of Proposition 3, all nodes keep their lists unchanged, therefore the results given by \text{EnumAll} for all nodes does not change from one iteration to the next one. This can be proven more formally with a basic induction over the number of recursive calls of \text{EnumAll}: the base case is a node with value \(\bot\); the inductive step is that, assuming the result for \(i\) recursions does not change, trivially the result for \(i + 1\) does not change either.

**Proposition 5.** Consider any state \(q\). For every iteration \(i\), a match \(M\) is in \(\mathcal{M}^i_q = \bigcup_{n \in \text{list}_q} \text{EnumAll}(n, n.\text{time})\) iff there is a run \(\rho\) of length \(i\) that ends in \(q\) and \(\text{match}(\rho) = M\).

**Proof.** This will be proven by induction over the number of iterations \(i\). As notation, we define \text{list}_i as the value of \text{list}_q at iteration \(i\). First, at \(i = 0\), all lists are empty, except for the initial states. Therefore, \(\mathcal{M}^0_q = \emptyset\) if \(q \notin I\) and \(\mathcal{M}^0_q = \{\emptyset\}\) if \(q \in I\). This is correct because in the former case there is no run, and in the latter case there is the run of length \(0\) with the empty match.

Now, for the inductive step consider that the proposition holds at iteration \(i\), and we want to show that it still holds at iteration \(i + 1\). First, we show the if direction. Consider a run of the form:

\[
\rho: q_0 \xrightarrow{\alpha_1/m_1} q_1 \xrightarrow{\alpha_2/m_2} \ldots \xrightarrow{\alpha_i/m_i} q_i \xrightarrow{\alpha_{i+1}/m_{i+1}} q_{i+1}
\]

With the associated match \(M\). By induction hypothesis, at iteration \(i\), \(M' \in \mathcal{M}^i_q\), with \(M' = M \setminus \{i + 1\}\) being the match of the run \(\rho\) up to the state \(q_i\). Now we see the two possible cases: if the last transition marked or not the last position.

In the former case, it must occur that there was a transition \((q_i, \alpha_{i+1}, \bullet, q_{i+1})\) such that the \((i + 1)\)-th event \(e_{i+1}\) satisfies \(\alpha_{i+1}\). Thus, when applying procedure \text{MoveMarking}(q_i, e_{i+1})\), the target node is \(q_{i+1}\) and a new node \(n\) with \(n.\text{time} = i + 1\) is added at the beginning of the new list \(\text{list}^i_{q_{i+1}}\). Moreover, because \(n.\top = \text{first}_{q_i}\) and \(n.\text{bot} = \text{last}_{q_i}\), it follows that \(\text{prev}_{n}\) is equal to \(\text{list}^i_{q_i}\) of the previous iteration. Here, it is important that Proposition 3 ensures that the value \(\text{prev}_{n}\) remains the same after doing iteration \(i + 1\). Notice that when applying procedure \text{EnumAll}(n, \cdot)\) the result
is exactly $\mathcal{M}^i_{q_i}$ and, by induction hypothesis, $M' \in \mathcal{M}^i_{q_i}$. Moreover, when applying procedure \texttt{EnumAll}(n, n.time), the result will contain all matches in $\mathcal{M}^i_{q_i}$ but with an extra position $n.time = i + 1$ at the end. Since, $M' \in \mathcal{M}^i_{q_i}$, it follows that $M' \cup \{i + 1\} = M \in \texttt{EnumAll}(n, n.time)$. Finally, since we know that $n \in \text{list}_{i+1}^{i+1}$, it holds that $M \in \texttt{EnumAll}(n, n.time) \subseteq \mathcal{M}^i_{q_i}$.

Now, consider the case that the last transition did not mark the position, i.e., $m_{i+1} = \circ$. We know that there must be a transition $(q_i, \alpha_{i+1}, \circ, q_{i+1})$ such that $e_{i+1} \equiv \alpha_{i+1}$. Then, when applying procedure \texttt{MoveNotMarking}(q_i, e_{i+1}), the target node is $q_{i+1}$ and the list $\text{list}_{i+1}^i$ is appended to the new list $\text{list}_{i+1}^i$. Therefore, $\text{list}_{i}^i \subseteq \text{list}_{i+1}^i$, thus $\mathcal{M}^i_{q_i} \subseteq \mathcal{M}^{i+1}_{q_{i+1}}$.

Because $M = M' \in \mathcal{M}^i_{q_i}$, it follows that $M \in \mathcal{M}^{i+1}_{q_{i+1}}$.

Now, we show the only-if direction. Assume that there is a match $M \in \mathcal{M}^{i+1}_{q_{i+1}}$. This means that there is some $n \in \text{list}_{i+1}^i$ such that $M \in \texttt{EnumAll}(n, n.time)$. Now, we consider two possible cases: if $M$ contains the position $i + 1$, or if it does not.

In the former case, it would mean that $n.time = i + 1$. Here, we consider the property that, for every node $n$, $n'.time < n.time$ for all $n' \in \text{prev}_n$. This should be clear from the algorithm, and can be formally proven by induction over the number of iterations in \texttt{Eval}. Then, the only possible case for this to happen is if the node $n$ was added in the last iteration (by the \texttt{MoveMarking} procedure). Thus, there must be some state $q_i$ and a transition $(q_i, \alpha_{i+1}, \bullet, q_{i+1})$ such that the $(i+1)$-th event $e_{i+1}$ satisfies $\alpha_{i+1}$. Moreover, $n$ was defined in a way that $\text{prev}_n = \text{list}_{i}^i$, thus $M' = M \setminus \{i + 1\} \in \mathcal{M}^{i}_{q_i}$. By induction hypothesis, this means that there is a run of the form:

$$\rho' : q_0 \xrightarrow{\alpha_{i+1}/m_{i+1}} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_i/m_i} q_i$$

Such that $\text{match}(\rho') = M'$. Moreover, by adding transition $(q_i, \alpha_{i+1}, \bullet, q_{i+1})$ at the end, we get:

$$\rho : q_0 \xrightarrow{\alpha_{i+1}/m_{i+1}} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_i/m_i} q_i \xrightarrow{\alpha_{i+1}/\bullet} q_{i+1}$$

Such that $\text{match}(\rho) = \text{match}(\rho') \cup \{i + 1\} = M' \cup \{i + 1\} = M$.

Consider now the latter case, that is, $i + 1 \notin M$. This means that $n$ got in $\text{list}_{i+1}^i$ because of the \texttt{MoveNotMarking} procedure. Thus, there must be some state $q_i$ and a transition $(q_i, \alpha_{i+1}, \bullet, q_{i+1})$ such that the $e_{i+1} \equiv \alpha_{i+1}$. Moreover, $n$ must have been in $\text{list}_{i}^i$, thus $M \in \mathcal{M}^{i}_{q_i}$. By induction hypothesis there must be a run:

$$\rho' : q_0 \xrightarrow{\alpha_{i+1}/m_{i+1}} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_i/m_i} q_i$$

Such that $\text{match}(\rho') = M$. By adding transition $(q_i, \alpha_{i+1}, \circ, q_{i+1})$ at the end, we get:

$$\rho : q_0 \xrightarrow{\alpha_{i+1}/m_{i+1}} q_1 \xrightarrow{\alpha_2/m_2} \cdots \xrightarrow{\alpha_i/m_i} q_i \xrightarrow{\alpha_{i+1}/\circ} q_{i+1}$$

Such that $\text{match}(\rho) = \text{match}(\rho') = M' = M$. \qed

Now, we have what we need to prove Theorem\textsuperscript{[10]} in particular, Proposition\textsuperscript{[5]} What \texttt{Enumerate} does at every iteration $i$ is just retrieve all $M \in \bigcup_{q \in F} \mathcal{M}_{q_i}^i$ such that $i \in M$. This last condition is necessary because $[A](S)$ only considers the matches that marked the last position $i$. Moreover, the procedure verifies it when considering only the nodes that have the current time $i$ in their \texttt{time} attribute. It is worth noting that all nodes with \texttt{time} attribute equal to $i$ are always at the top of the list, because they were appended at the top in iteration $i$ by \texttt{MoveMarking}.

Formally, we know that at iteration $i$, \texttt{Enumerate}(i) enumerates a match $M$ iff there is a run of the form:

$$\rho : q_0 \xrightarrow{\alpha_0/m_0} q_1 \xrightarrow{\alpha_1/m_1} \cdots \xrightarrow{\alpha_i/m_i} q_i$$

Such that $\text{match}(\rho) = M$ and $q_i \in F$. Moreover, we know that $\alpha_i = \bullet$, therefore $M \in [A](S)$. For the other direction, if $M \in [A](S)$, then $\rho$ must exist. Thus, by Proposition\textsuperscript{[5]}, \texttt{Enumerate}(i) enumerates $M$. \qed

It is left to prove that \texttt{Enumerate}(\texttt{time}) can run with constant delay. For this, we provide Algorithm\textsuperscript{[3]} which does the same as \texttt{EnumAll} and runs with constant delay. Moreover, the algorithm takes constant time between each position of each match, and constant time between matches.

The notation in Algorithm\textsuperscript{[3]} is explained now: \texttt{match_stack} and \texttt{node_stack} are stacks, which provide the traditional methods \texttt{push}(), \texttt{pop}() and \texttt{pop}() (the latter takes out the last $i$ elements); \texttt{enum(}) retrieves to the user the position $p$ or a set of positions in $p$ (if $p$ is a position or a stack, respectively). Finally, we assume that if $n = \bot$ then $n.time$ contains some separator symbol (e.g. #) so that, at the end of the algorithm, the enumerated elements have a structure like $M_1 \# M_2 \# \ldots \# M_k$.

The idea of \texttt{EnumAll}\textsuperscript{'} is the following. In order to keep the explanation simple, we will see the data structure that stores the matches as a tree, where each node $n$ has the nodes of $\text{prev}_n$ as its children (in practice this is not the case since some nodes can share the same children, but this will not affect the procedure). Both \texttt{EnumAll} and \texttt{EnumAll}\textsuperscript{'} are based on the same intuition: to navigate through the tree in a Depth First Search manner and compute a match for each path from the root to a leaf. The main difference is that, while \texttt{EnumAll} does this with recursion and moves one node at a time, \texttt{EnumAll}\textsuperscript{'} can move up an arbitrary number of nodes when it acknowledges that there are no
more matches at that section of the tree. This is particularly useful in cases when, for example, the tree consists of only two disjoint paths that meet at the root. In this scenario, after enumerating the match $M_1$ of the first path, EnumAll would have to go back to the root through $|M_1|$ nodes before enumerating the match $M_2$ for the second path, thus taking time $O(|M_1|)$ between $M_1$ and $M_2$. On the other side, EnumAll uses a stack $n_{stack}$ to store the exact node at which it has to go back (in this case, the root node), therefore it takes constant time between matches. Moreover, to store the partial match, EnumAll uses a stack $match_{stack}$ which adds each position when going down the tree and deletes the corresponding part when jumping up. This way, EnumAll ensures that the time it takes between enumerating each position is constant.

Finally, one can see that Algorithm 1 using procedure EnumAll results into an $f$-evaluation strategy for $A$.  

\section{Proof of Theorem 8}

In Algorithm 4 we provide an efficient strategy to evaluate a MA over the $\text{NXT}$ semantics. Similar to the MA of the $\text{NXT}$ construction, we need to keep track of the order of priority between all runs. For this, the algorithm stores the runs in a queue structure $E$, which has the functions $\text{enqueue}$, to add a new element at the end, and $\text{dequeue}$, to extract the first element. Here, each element in $E$ represents a tuple $(T, M)$ where $T$ is a non-empty set of states and $M$ is a match. We use the function $\text{notin}$ which receives a state $q$ and a queue $E$, and has the value TRUE if $q$ does not appear in any run of $E$, and FALSE otherwise.

For all $(T, M)$ in $E$, every $q \in T$ represents a run of $A$ whose associated partial match is precisely $M$. To this end, the subroutine Update computes the set $T'$ of states that can be reached from all the states of $T$ using a transition with mark $m$. Afterwards, it adds the new tuple $(T', M')$ to $E'$, where $M'$ is equal to $M$ if $m = \circ$, and is equal to $M$ plus the position $i$ currently read if $m = \bullet$. After applying Update over all the elements of $E$, the resulting set $E'$ is similar to the result of the function $\text{TPO}$ in Section 10.3.2 of Theorem 8 in the sense that the result only stores information about the runs that could lead to a match of the $\text{NXT}$ semantics. The key to achieve this is that the higher runs are updated first (line 5) and, moreover, the update is first done using $\bullet$ transitions, and later using $\circ$ (lines 6 and 7). Therefore, at each iteration the queue $E = [(T_0, M_0), \ldots, (T_n, M_n)]$ is updated in such a way that:

\begin{itemize}
\item The matches follow the (reversed) $<_{\text{next}}$ order, i.e., $M_n <_{\text{next}} M_{n-1} <_{\text{next}} \ldots <_{\text{next}} M_0$.
\item No state can appear twice in $E$, and
\item For every $q \in T_i$, the match $M_i$ is the highest of all the partial runs of $A$ that could currently be in $q$.
\end{itemize}

Finally, after updating $E$ the algorithm retrieves the highest match $M$ (w.r.t. the $<_{\text{next}}$ order) if there exists a run associated to $M$ currently in a final state $q$.

It is clear to see that Algorithm 4 iterates over all states of $A$ (because each one appears at most one in $E$), and for each one it iterates over all transitions that begin at $q$. Therefore, if we assume that the transitions are indexed by their initial states, it is easy to see that the iteration for each event takes linear time over the size of $A$. □
Algorithm 4 Evaluate $\mathcal{A} = (Q, \Delta, I, F)$ over a stream $S$ with $\text{NXT}$ semantics

1: procedure $\text{Eval}[\mathcal{A}](S)$
2: \hspace{1em} $E \leftarrow [(I, \varnothing)]$
3: \hspace{1em} while $t \leftarrow \text{yield}_S$ do
4: \hspace{2em} $E' \leftarrow []$
5: \hspace{2em} for $(T, M) \leftarrow E.\text{dequeue}()$ do
6: \hspace{3em} $E' \leftarrow \text{Update}(E', T, M, t, \bullet)$
7: \hspace{3em} $E' \leftarrow \text{Update}(E', T, M, t, \circ)$
8: \hspace{2em} end for
9: \hspace{1em} $E \leftarrow E'$
10: \hspace{1em} enumerate($\text{arg min}_M \{ j \mid (T, M) = E[j] \land F \cap T \neq \emptyset \}$)
11: end while
12: end procedure

13: procedure $\text{Update}[\mathcal{A}](E', T, M, t, m)$
14: \hspace{1em} $T' \leftarrow \emptyset$
15: \hspace{1em} for all $q \in T \land (q, \alpha, m, q') \in \Delta$ do
16: \hspace{2em} if $t \models \alpha \land \text{notin}(q, E')$ then
17: \hspace{3em} $T' \leftarrow T' \cup \{ q' \}$
18: \hspace{2em} end if
19: end for
20: if $T' \neq \emptyset$ then
21: \hspace{1em} if $m = \bullet$ then
22: \hspace{2em} $E'.\text{enqueue}((T', M \cup \{ i \}))$
23: \hspace{2em} else
24: \hspace{2em} $E'.\text{enqueue}((T', M))$
25: \hspace{2em} end if
26: end if
27: return $E'$
28: end procedure