Complex Event Recognition with Symbolic Register Transducers

Elias Alevizos
NCSR “Demokritos”
alevizos.elias@iit.demokritos.gr

Alexander Artikis
Univ. of Piraeus, NCSR “Demokritos”
a.artikis@unipi.gr

Georgios Paliouras
NCSR “Demokritos”
paliourg@iit.demokritos.gr

ABSTRACT

We present a system for Complex Event Recognition (CER) based on automata. While multiple such systems have been described in the literature, they typically suffer from a lack of clear and denotational semantics, a limitation which often leads to confusion with respect to their expressive power. In order to address this issue, our system is based on an automaton model which is a combination of symbolic and register automata. We extend previous work on these types of automata, in order to construct a formalism with clear semantics and a corresponding automaton model whose properties can be formally investigated. We call such automata Symbolic Register Transducers (*SRT*). The distinctive feature of *SRT*, compared to previous automaton models used in CER, is that they can encode patterns relating multiple input events from an event stream, without sacrificing rigor and clarity. We show how *SRT* can be used in CER in order to detect patterns upon streams of events, using our framework that provides declarative and compositional semantics. We also compare our *SRT*-based CER engine against other state-of-the-art CER systems and show that it is both more expressive and more efficient.

PVLDB Reference Format:
Elias Alevizos, Alexander Artikis, and Georgios Paliouras. Complex Event Recognition with Symbolic Register Transducers. PVLDB, 17(11): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ElAlev/cer-srt.

1 INTRODUCTION

A Complex Event Recognition (CER) system takes as input a stream of “simple events”, along with a set of patterns, defining relations among the input events, and detects instances of pattern satisfaction, thus producing an output stream of “complex events” [16, 25, 34]. For simplicity, we will henceforth refer to “simple events” as “events” and “complex events” as “patterns”. Typically, an event takes the form of a tuple comprising numerical or categorical values. Complex events must often be detected with very low latency [23, 29, 34]. Table 1 presents a simple example of a stream of event tuples.

Automata are of particular interest for the field of CER, because they provide a natural way of handling sequences. As a result, the usual operators of regular expressions, like concatenation, union and Kleene-star, have often been given an implicit temporal interpretation in CER. For example, the concatenation of two events is said to occur whenever the second event is read by an automaton after the first one, i.e., whenever the timestamp of the second event is greater than the timestamp of the first. On the other hand, atemporal constraints are not easy to define using classical automata, since they either work without memory or, even if they do include a memory structure, e.g., as with push-down automata, they can only work with a finite alphabet of input symbols. For this reason, the CER community has proposed several extensions of classical automata. These extended automata have the ability to store input events and later retrieve them in order to evaluate whether a constraint is satisfied [8, 16, 21]. They resemble both register automata [30], through their ability to store events, and symbolic automata [18], through the use of predicates on their transitions. They differ from symbolic automata in that predicates apply to multiple events, retrieved from the memory structure that holds previous events. They differ from register automata in that predicates may be more complex than that of (in)equality.

One issue with these CER-specific automata is that their properties have not been systematically investigated, in contrast to models derived directly from the field of languages and automata; see [26] for a discussion about the weaknesses of automaton models in CER. Moreover, they sometimes need to impose restrictions on the use of regular expression operators in a pattern, e.g., nesting of Kleene-star operators is not allowed. We propose a system for CER, based on an automaton model which can address these issues. This model is a combination of symbolic and register automata. We call such automata *Symbolic Register Transducers* (*SRT*). *SRT* extend the expressive power of symbolic and register automata, by allowing for more complex patterns to be defined and detected on a stream of events. We also present a language with which we can define patterns for complex events that can then be translated to *SRT*. We call such patterns *Symbolic Regular Expressions with Memory and Output* (*SREMO*), as an extension of the work presented in [32], where *Regular Expressions with Memory* (*REM*) are defined and investigated. *REM* are extensions of classical regular expressions with which some of the terminal symbols of an expression can be stored and later be compared for (in)equality. *SREMO* allow for more complex conditions to be used, besides those of (in)equality. They additionally allow each terminal sub-expression to mark an element as belonging or not to the string/match that is to be recognized, thus acting as transducers.

Our contributions may then be summarized as follows:

- We present a CER system based on a formal framework with denotational and compositional semantics, where patterns may be written as *SREMO*.
- We show how this framework subsumes, in terms of expressive power, previous similar attempts. It allows for nesting

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: Example of a stream of stock market ticks. A stream is a sequence of events, where each such event is a tuple of the form (*type*, *id*, *price*, *volume*). *type* is the type of transaction: *S* for SELL and *B* for BUY. *id* is an integer identifier, unique for each company. It has a finite set of possible values. *price* is a real-valued number for the price of a given stock. Finally, *volume* is a natural number referring to the volume of the transaction. Events are assumed to be temporally ordered and their order is implicitly provided through the index. Concurrent events cannot occur, i.e., each index is unique to a single event.

type	B	B	B	S	S	B	...
id	1	1	2	1	1	2	...
price	22	24	32	70	68	33	...
volume	300	225	1210	760	2000	95	...
index	1	2	3	4	5	6	...

operators and selection strategies. It also allows n-ary expressions to be used as conditions in patterns, thus allowing the detection of relational patterns.

- We extend previous work on automata and present a computational model for patterns written in *SREMO*, Symbolic Register Transducers (*SRT*), whose main feature is that it supports relations between multiple events. *SRT* also have the ability to mark exactly those events comprising a match.
- We show that *SRT* are closed under the most common operators, i.e., union, intersection, concatenation and Kleene-star. Moreover, we show that, by using windows, *SRT* are closed under complement and determinization. Windows are an indispensable operator in CER because, among others, they limit the search space for pattern matching.
- We implement a CER engine with *SRT* at its core and present relevant experimental results. Our engine is both more efficient than other engines and supports a language that is more expressive than that of other systems.

All proofs and complete algorithms may be found in an extended technical report[1]. In Table 2 we have gathered the notation that we use throughout the paper.

2 RELATED WORK

Typical cases of CER systems that employ automata are the Chronicle Recognition System [22, 24], Cayuga [20, 21], TESLA [15], SASE [8, 45], CORE [13, 26] and Wayeb [9, 10]. There also exist systems that do not employ automata as their computational model, e.g., there are logic-based systems [35, 40] or systems that use trees [36], but the standard operators of concatenation, union and Kleene-star are quite common and they may be considered as a reasonable set of core operators for CER. The abundance of different CER systems, employing various computational models and using various formalisms has recently led to some attempts to provide a unifying framework [26, 28]. In [26], a set of core CER operators is identified, a formal framework is proposed that provides denotational semantics for CER patterns, and a computational model is described for capturing such patterns. For an overview of CER languages, see [25], and for a general review of CER systems, see [16]. In this Section, we present previous related work along three axes. First, we discuss previous theoretical work on automata that is related to CER. We subsequently present previous automata-based CER systems. Finally, we briefly discuss some solutions which are beyond the scope of CER in the strict sense of the term, but have characteristics that are of interest to CER. Table 3 summarizes our discussion and provides a compact way to compare our proposal against previous solutions.

Table 2: Notation used throughout the paper.

Symbol	**Meaning**
$\mathcal{U}/\mathcal{L} \subseteq \mathcal{U}^*/t_i \in \mathcal{U}$	universe/language over $\mathcal{U}$/character
$S = t_1, t_2, \cdots$ / $S_{i..j} = t_i, \cdots, t_j$	stream / stream "slice" from index i to j
$R = \{r_1, \cdots, r_k\}$	register variables
$v : R \hookrightarrow \mathcal{U}$	valuation
$\sharp$ / $\sim$	contents of empty register / automaton head
$(u, v) \models \phi$	condition ϕ satisfied by element u and valuation v
$\bullet, \otimes$	outputs
$e_1 + e_2$ / $e_1 \cdot e_2$ / e^* / $@e$	regular disjunction / concatenation / iteration / skip-till-any-match
$e^{[1..w]}$	windowed expression with window size w
$(e, S, M, v) \vdash v'$	string S and match M on expression e with valuation v induce valuation v'
$Lang(e)/Match(e, S)$	language/matches of expression e
$c = [j, q, v]$	configuration (position, state, valuation)
$Lang(T)/Match(T, S)$	language/matches of automaton T

Extended automaton models: theory. Outside the field of CER, research on automata has evolved towards various directions. Besides push-down automata that can store elements from a finite set to a stack, there are other automaton models with memory, such as register automata, pebble automata and data automata [11, 30, 37]. For a review, see [39]. Such models are especially useful when the input alphabet cannot be assumed to be finite, as is often the case with CER. Register automata (initially called finite-memory automata) constitute one of the earliest such proposals [30]. At each transition, a register automaton may choose to store its current input (more precisely, the current input's data payload) to one of a finite set of registers. A transition is followed if the current input is equal to the contents of some register. With register automata, it is possible to recognize strings constructed from an infinite alphabet, through the use of (in)equality comparisons among the data carried by the current input and the data stored in the registers. However, register automata do not always have nice closure properties, e.g., they are not closed under determinization. For an extensive study of register automata, see [32, 33]. We build on the framework presented in [32, 33] to construct register automata with the ability to handle "arbitrary" structures, besides those containing only (in)equality relations. Another model that is of interest for CER is the symbolic automaton, which allows CER patterns to apply constraints on the

[1] https://arxiv.org/abs/2407.02884

System	σ_1	σ_n	$\vee$	$\wedge$	!	;	*	D	E	S.P.	Remarks
Theory											
Register automata	✗	✗	✓	✓	✗	✓	✓	✗	✗	Sc	Selection only for unary (in-)equality.
Symbolic automata	✓	✗	✓	✓	✓	✓	✓	✓	✗	Sc	
Symbolic register automata	✓	✓	✓	✓	✗	✓	✓	✗	✗	Sc	
Automata-based CER solutions											
SASE	✓	✓	✗	✗	✓	✓	✓	✗	✓	all	Iteration and selection strategies cannot be nested. $\vee$, $\wedge$ and ¬ possible in principle but not available in source code. Soundness issues with selection strategies
Cayuga	✓	✓	✓	?	✗	✓	✓	✗	✗	Stam	Re-subscription with multiple automata for nested expressions.
FlinkCEP	✓	✓	✓	?	✓	✓	?	✗	✓	?	Soundness issues with selection strategies and iteration.
Esper	✓	✓	✓	?	✓	✓	✓	?	✓	all	Mixture of trees, automata and Allen's interval algebra.
CORE	✓	✗	✓	?	?	✓	✓	✓	✓	all	
Wayeb (symbolic automata)	✓	✗	✓	✓	✓	✓	✓	✓	✗	all	
Beyond CER											
AFA	✓	?	✓	?	?	✓	✓	?	✗	Sc	Partial support of negation. σ_n with a single register.
MATCH_RECOGNIZE	✓	✓	✗	?	✓	✓	✗	?	✗	all	Supported features depend on the implementation.
Our proposal											
Wayeb (SRT)	✓	✓	✓	✓	✓	✓	✓	✓	✓	all	¬ and determinization supported only for windowed expressions.

Table 3: Comparing state-of-the-art with our proposal. σ_1: unary selection, σ_n: n-ary selection, $\wedge$: intersection, $\vee$: union, !: negation, ;: sequence, *: iteration, D: determinizability, E: enumeration, S.P.: selection strategies, Stam : skip-till-any-match, Stnm : skip-till-next-match, Sc : strict-contiguity.

attributes of events. Automata that have predicates on their transitions were already proposed in [41]. This initial idea has recently been expanded and more fully investigated in symbolic automata [18, 42, 43]. In symbolic automata, transitions are equipped with formulas constructed from a Boolean algebra. A transition is followed if its formula, applied to the current input, evaluates to TRUE. Contrary to register automata, symbolic automata have nice closure properties, but their formulas are unary and thus can only be applied to a single element from the input string. This is one limitation that we address here. We use *Symbolic Regular Expressions with Memory and Output* (*SREMO*) and *Symbolic Register Transducers* (*SRT*), a language and an automaton model respectively, that can handle n-ary formulas and be applied for the purposes of CER. With *SREMO* we can designate which elements of a pattern need to be stored for later evaluation and which must be marked as being part of a match. *SREMO* can be compiled into *SRT*, whose transitions can apply n-ary formulas/conditions (with n>1) on multiple elements. As a result, *SRT* are more expressive than symbolic and register automata, thus being suitable for practical CER applications, while, at the same time, their properties can be systematically investigated, as in standard automata theory. In fact, our model subsumes these two automaton models as special cases. It is also an extension of Symbolic Register Automata [17], which do not have any output on their transitions and cannot thus enumerate the detected complex events, since they do not have the ability to mark input events as being part of match. Moreover, the applicability of *SRT* for CER is studied here for the first time. We show precisely how *SRT* can be used for CER and how the use of *SRT* provides expressive power without sacrificing clarity and rigor.

Extended automaton models as applied in CER. Automata with registers have been proposed in the past for CER, e.g., in SASE and Cayuga. However, previous systems typically provide operational semantics and it is not always clear a) what operators are allowed, b) at which combinations, c) what the properties of their automaton models are. For example, SASE's language seems to support nested Kleene operators. However, it constructs automata whose states are linearly ordered. Therefore, Kleene operators can only be applied to single states. They cannot be nested and they cannot contain other expressions, except for single events. As a result, disjunction is also not allowed. On the other hand, Cayuga attempts to address these expressivity limitations through the method of resubscription, i.e., expressions which cannot be captured by a single automaton are compiled into multiple automata [19]. Each sub-automaton can then subscribe to the output of other automata, thus creating a hierarchy of automata. Although this is an interesting solution, the resulting semantics remains ambiguous, since the correctness and limits of this approach have not been thoroughly investigated. The novelty of our system is that it provides formal, compositional semantics which allows us to address all of the above issues. With the exception of negation, all other operators may be arbitrarily combined in a completely compositional manner and each pattern can be compiled into a single automaton, something which has not been previously achieved. CORE [26, 27] and Wayeb [9, 10] constitute two more recent automata-based CER systems. CORE automata may be categorized under the class of "unary" symbolic automata (or transducers, to be more precise), i.e., they do not support patterns relating multiple events. The same is true for Wayeb, which also employs "unary" symbolic automata.

Extended automaton models beyond CER. An adaptation of finite automata in the context of Data Stream Management Systems (which have strong similarities to CER systems) has also been proposed in [14]. These automata are called augmented finite automata (AFA) and are enriched with registers, in order to capture trends. Like *SRT*, they support arbitrary edges and are compositional. On the other hand, AFA have different limitations. Each AFA has a single register (one per active state), whereas there is no such restriction for *SRT*. Additionally, AFA are not transducers and cannot enumerate the input events of a complex event. Finally, the properties of AFA have not been theoretically studied, for example with respect to determinization and negation. AFA can handle certain instances of negation, but there are strong reasons to suspect that they are not in general closed under complement, as is the case of register automata. In summary, *SRT* are more expressive than AFA. Another way to implement CER patterns, in relational databases, is through SQL's MATCH_RECOGNIZE, a proposed clause that can perform pattern recognition on rows [6, 38]. MATCH_RECOGNIZE is very expressive and can in principle capture almost any pattern expressed in a CER language. However, it is uncertain whether it would work in a streaming setting as efficiently as CER systems. Recent work has proposed implementations of MATCH_RECOGNIZE that are more efficient than the one already available in Flink [31, 46]. The proposed optimizations rely on the use of prefiltering and clever indices so that the automaton responsible for pattern recognition is fed only with a small subset of the initial rows. They target the scenario of historical analysis and their extension to a streaming setting is not considered. It still remains an open issue whether the proposed optimizations would work for patterns processing events in real time.

3 SYMBOLIC REGULAR EXPRESSIONS WITH MEMORY AND OUTPUT

We start by presenting a language for CER and discuss its semantics. The main feature of this language is that it allows for most of the common CER operators (such as selection, sequence, disjunction and iteration), without imposing restrictions on how they may be used and nested. Our proposed language can also accommodate n-ary conditions, i.e., we can impose constraints on the patterns which relate multiple events of a stream, e.g., that the price of a stock at the current timepoint is higher than its price at the previous timepoint. We also discuss the semantics of patterns written in the proposed language and show that these are well-defined. Hence, in order to know whether a given stream contains complex events corresponding to a given pattern, we do not need to resort to a procedural computational model. The semantics of the language may be studied independently of the chosen computational model. This feature is critical, as it allows for a systematic understanding of the use of operators. Additionally, it could be of importance for optimization, which often relies on pattern re-writing, assuming that we can know when two patterns are equivalent without actually having to run their computational models.

We extend the work presented in [32], where the notion of regular expressions with memory (*REM*) was introduced. These regular expressions can store some terminal symbols, in order to compare them later (in a string) against a new input element for (in)equality. The corresponding automata compiled from *REM* need to be equipped with registers. Each transition has the option to write the symbol that triggered it to some register. Transitions can also access registers to retrieve their contents (previously stored elements) and compare them with the last element read by the automaton's head. One important limitation of *REM* with respect to CER is that they can handle only (in)equality relations. In this section, we extend *REM* so as to endow them with the capacity to use relations from "arbitrary" structures. We call these extended *REM* *Symbolic Regular Expressions with Memory and Output* (*SREMO*).

We assume that each terminal expression of a *SREMO* is a Boolean expression whose predicates are in the form of a relation P. We also assume that all possible input events constitute a universe $\mathcal{U}$ (details may be found in the technical report). We can then extend the terminology of classical regular expressions to define characters, strings and languages. Elements of $\mathcal{U}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathcal{L}$ constructed from elements of $\mathcal{U}$, i.e., $\mathcal{L} \subseteq \mathcal{U}^*$, where * denotes Kleene-star, is called a language over $\mathcal{U}$. Then, a stream S is an infinite sequence $S = t_1, t_2, \cdots$, where each $t_i \in \mathcal{U}$ is a character. By $S_{1..k}$ we denote the sub-string of S composed of the first k elements of S. $S_{m..k}$ denotes the slice of S starting from the m^{th} and ending at the k^{th} element. We can then define n-ary relations P on the elements of $\mathcal{U}$ and use these relations, or combinations of them via Boolean connectives, as terminal expressions within a regular expression. The arguments of P refer either to the most recently read element of a string or to preceding elements, assumed to have been stored in registers. We call such terminal expressions "conditions". Conditions are the basic building blocks of *SREMO*. In the simplest case, they are applied to single events and act as filters. In the general case, we need them to be applied to multiple events, some of which may be stored to registers. Conditions will essentially be the n-ary guards on the transitions of *SRT*.

Definition 3.1 (Condition). Let $\top$ be a unary relation for which it holds that $u \in \top, \forall u \in \mathcal{U}$, i.e., this relation holds for all elements of the universe $\mathcal{U}$. Let $R = \{r_1, \cdots, r_k\}$ be variables denoting the registers and $\sim$ a special variable denoting an automaton's head which reads new elements. The "contents" of the head always correspond to the most recent element. We call R register variables. A condition is then defined by the following grammar:

- $\top$ is a condition.
- $P(r_1, \cdots, r_n)$, where $r_i \in R \cup \{\sim\}$ and P an n-ary relation, is a condition.
- $\neg\phi$ is a condition, if ϕ is a condition.
- $\phi_1 \wedge \phi_2$ is a condition if ϕ_1 and ϕ_2 are conditions.
- $\phi_1 \vee \phi_2$ is a condition if ϕ_1 and ϕ_2 are conditions. ◄

Example 3.2. As an example, consider the simple case where we want to detect stock ticks of type BUY (B), followed by a tick of type SELL (S) for the same company. We would thus need a simple condition on the first tick, denoted as $TypeIsB(\sim)$, where $TypeIsB(x){:=}x.type{=}B$. $TypeIsB(\sim)$ has a single argument, the automaton head. We also need another condition for the SELL tick and the company comparison, denoted as $TypeIsS(\sim) \wedge EqualId(\sim, r_1)$. We assume that $TypeIsS(x){:=}x.type{=}S$ and $EqualId(x, y){:=}x.id{=}y.id$. Note that, beyond the head variable, *EqualId* also has a register variable as an input argument. ◄

Note that the arguments of P in Definition 3.1 refer either to registers or to the current element of the head. We thus need a way to conceptualize how the contents of these registers may be accessed and modified. The notion of valuations serves this purpose.

Definition 3.3 (Valuation). Let $R = \{r_1, \cdots, r_k\}$ be a set of register variables. A valuation on R is a partial function $v : R \hookrightarrow \mathcal{U}$, i.e., some registers may be "empty". We will also use the notation $v(r_i){=}\sharp$ to denote the fact that register r_i is empty, i.e., we extend the range of v to $\mathcal{U} \cup \{\sharp\}$. We also extend the domain of v to $R \cup \{\sim\}$.

By $v(\sim)$ we denote the "contents" of the automaton's head, i.e., the last element read from the string. ◀

For the base case of condition $\phi := P(r_1, \cdots, r_n)$, we say that ϕ is satisfied by (u, v), denoted by $(u, v) \models \phi$, iff $u \in P(v(x_1), \cdots, v(x_n))$. For the remaining cases, the semantics of conditions are defined as in standard Boolean expressions.

We can now define *SREMO*, by combining conditions via the standard regular operators. Conditions act as terminal expressions, i.e., the base case upon which we construct more complex expressions. Each condition may be accompanied by a register variable, indicating that an event satisfying the condition must be written to that register. It may also be accompanied by an output, either •, indicating that the event must be marked as being part of the complex event, or ⊗, indicating that the event is irrelevant and should be excluded from any detected complex events.

Definition 3.4 (Symbolic regular expression with memory and output (SREMO)). A *SREMO* is inductively defined as follows:

(1) ϵ is a *SREMO*.
(2) If ϕ is a condition (as in Definition 3.1) and $o \in \{\bullet, \otimes\}$ an output, then $\phi \uparrow o$ is a *SREMO*.
(3) If ϕ is a condition, $o \in \{\bullet, \otimes\}$ an output and r_i a register variable, then $\phi \uparrow o \downarrow r_i$ is a *SREMO*.
(4) If e_1 and e_2 are *SREMO*, then $e_1 + e_2$ is also a *SREMO*.
(5) If e_1 and e_2 are *SREMO*, then $e_1 \cdot e_2$ is also a *SREMO*.
(6) If e is a *SREMO*, then e^* is also a *SREMO*. ◀

ϵ is the regular expression satisfied by the "empty" string, i.e., without any characters. With *SREMO* of the form $\phi \uparrow o \downarrow r_i$ (case 3 above), we denote cases where we need to store the current element read from the automaton's head to register r_i. If we additionally need to mark the event as part of the match, we write $o = \bullet$. We write $o = \otimes$ when we do not want to mark the current element. Case 4 corresponds to the usual disjunction, whereas case 5 to concatenation. Finally, case 6 is the Kleene-star operator. Disjunction, concatenation and Kleene-star are the three standard operators in regular expressions which are also used here.

Example 3.5. We now have everything we need to express the pattern of our example. Consider the following *SREMO*:

$$\begin{aligned} e_1 :=& (TypeIsB(\sim) \uparrow \bullet \downarrow r_1) \cdot (\top \uparrow \otimes)^* \cdot \\ & ((TypeIsS(\sim) \wedge EqualId(\sim, r_1)) \uparrow \bullet) \end{aligned} \tag{1}$$

e_1 first looks for elements of type BUY. When it finds one, it marks it as belonging to a (candidate match) and writes it to register r_1. r_1 stores the whole element. For example, if e_1 starts processing the stream of Table 1, after reading the first element, r_1 will have stored $(B, 1, 22, 300)$. e_1 can then skip any number of elements, without marking or storing them, until encountering a SELL element from the same company. It marks it as part of the match as well. ◀

In order to define the semantics of *SREMO*, we begin with a *SREMO* e and a string S. Due to space limitations, the formal definition for the semantics of *SREMO* is presented in the technical report. Here we only make some brief remarks. In automata theory, we typically want to determine whether S is accepted by e. In our case, the situation is more complex. The goal is actually twofold: a) determine whether e can start from an empty valuation (all registers are empty) and follow a "path" leading to the end of S; b) determine whether such a "path" marks any events and thus generates a match. We thus need to first define how a *SREMO*, starting from a given valuation v and reading a given string S, reaches another valuation v'. Our final aim is to detect matches of a *SREMO* e in a string $S{=}t_1, \cdots, t_n$. A match $M{=}\{i_1, \cdots, i_k\}$ of e on S is a totally ordered set of natural numbers, referring to indices in the string S, i.e., $i_1 \geq 1$ and $i_k \leq n$ (see also [13]). If $M{=}\{i_1, \cdots, i_k\}$ is a match of e on S, then the set of elements referenced by M, $S[M]{=}\{t_{i_1}, \cdots, t_{i_k}\}$ represents a *complex event*. We write $M = M_1 \cdot M_2$ for two matches M_1, M_2 to denote the fact that $M_1 \cap M_2 = \emptyset$, $M_1 \cup M_2 = M$ and $max(M_1) < min(M_2)$. With the notation $(e, S, M, v) \vdash v'$, we denote that a *SREMO* e, starting from a valuation v, can read a string S and produce a match M, while reaching a new valuation v'.

Based on the above, we may now define the language that a *SREMO* accepts and the matches that it detects on a string S. The language of a *SREMO* contains all the strings with which we can reach a valuation, starting from the empty valuation, where all registers are empty. The set of matches is composed of all the matches computed after a *SREMO* has processed a string S.

Definition 3.6 (Language accepted and matches detected by a SREMO). The language accepted by a *SREMO* e is defined as

$$Lang(e) = \{S \mid (e, S, M, \sharp) \vdash v\}$$

for some valuation v and some match M of e on the corresponding S, where $\sharp$ denotes the valuation in which no $v(r_i)$ is defined, i.e., all registers are empty. The matches detected by a *SREMO* e on a string S is defined as

$$Match(e, S) = \{M \mid (e, S, M, \sharp) \vdash v\}$$

for some valuation v. ◀

Example 3.7. We can now continue with our example. If we feed the string/stream of Table 1 to *SREMO* (1), then we see that $S_{1..4}$ is indeed accepted by e_1. $M = \{1, 4\}$ is also a match of e_1 on S (and on $S_{1..4}$). The same is also true for $S_{1..5}$ and $M = \{1, 5\}$. ◀

The above introduction highlights the expressiveness, flexibility and formal semantics of *SREMO*. *SREMO* can express relational patterns with n-ary constraints, by being able to relate the most recently read element with any of the preceding ones. They also allow for arbitrary nesting of the regular operators, without imposing ad hoc restrictions. Moreover, their expressive power is combined with clear, denotational semantics.

4 SYMBOLIC REGISTER TRANSDUCERS

In order to capture *SREMO*, we propose Symbolic Register Transducers (*SRT*), an automaton model equipped with memory, logical conditions on its transitions and a single output on every transition. The basic idea is the following. We add a set of registers R to an automaton in order to be able to store events from the stream that will be used later in n-ary formulas. Each register can store at most one event. In order to evaluate whether to follow a transition or not, each transition is equipped with a guard, in the form of a Boolean formula. If the formula evaluates to TRUE, then the transition is followed. Since a formula might be n-ary, with $n{>}1$, the values passed

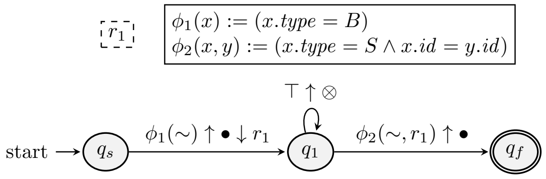
r_1
$\phi_1(x) := (x.type = B)$
$\phi_2(x, y) := (x.type = S \wedge x.id = y.id)$
$\top \uparrow \otimes$
start
q_s
$\phi_1(\sim) \uparrow \bullet \downarrow r_1$
q_1
$\phi_2(\sim, r_1) \uparrow \bullet$
q_f

Figure 1: *SRT* corresponding to the *SREMO* of eq. (1).

to its arguments during evaluation may be either the current event or the contents of some registers, i.e., some past events. In other words, the transition is also equipped with a *register selection*. Before evaluation, the automaton reads the contents of the required registers, passes them as arguments to the formula and the formula is evaluated. Additionally, if, during a run of the automaton, a transition is followed, then the transition has the option to write the event that triggered it to some of the automaton's registers. These are called its *write registers* W, i.e., the registers whose contents may be changed by the transition. Finally, each transition, when followed, produces an output, either $\otimes$, denoting that the event is not part of the match for the pattern that the *SRT* tries to capture, or $\bullet$, denoting that the event is part of the match. We also allow for ϵ-transitions, as in classical automata, i.e., transitions that are followed without consuming any events and without altering the contents of the registers.

We now formally define *SRT*. To aid understanding, we present three separate definitions. The first concerns the automaton itself, describing its structure, i.e., its states and transitions. The remaining two describe the running behavior of a *SRT*. For this we need to know its current state and register contents after every new event, i.e., its so-called configuration. We also need to know how the automaton changes configurations and how such a succession of configurations (a so-called run) may lead to a match.

Definition 4.1 (Symbolic Register Transducer). A symbolic register transducer (*SRT*) with k registers is a tuple (Q, q_s, Q_f, R, Δ) where

- Q is a finite set of states,
- $q_s \in Q$ the start state,
- $Q_f \subseteq Q$ the set of final states,
- $R = (r_1, \cdots, r_k)$ a finite set of registers and
- Δ the set of transitions.

A transition $\delta \in \Delta$ is a tuple (q, ϕ, W, q', o), also written as $q, \phi{\uparrow}o{\downarrow}W{\rightarrow}q'$, where

- $q, q' \in Q$, where q is the source and q' the target state,
- ϕ is a condition, as per Definition 3.1 or $\phi = \epsilon$,
- $W \in 2^R$ are the write registers and
- $o \in \{\otimes, \bullet\}$ is the output. ◄

We will use the dot notation to refer to elements of tuples. For example, if T is a *SRT*, then $T.Q$ is the set of its states. For a transition δ, we will also use the notation $\delta.source$ and $\delta.target$ to refer to its source and target states respectively.

Example 4.2. As an example, consider the *SRT* of Figure 1. Each transition is represented as $\phi \uparrow o \downarrow W$, where ϕ is its condition, o its output and W its set of write registers (or simply r_i if only a single register is written). W may also be an empty set, implying that no register is written. In this case, we avoid writing W on the transition (see, for example, the transition from q_1 to q_f in Figure 1). o may be omitted, in which case it is implicitly assumed that $o = \otimes$. The definitions for the conditions of the transitions are presented in a separate box, above the *SRT*. Note that the arguments of the conditions correspond to registers, through the register selection. Take the transition from q_s to q_1 as an example. It takes the last element consumed from the string/stream ($\sim$) and passes it as argument to the unary formula ϕ_1. If ϕ_1 evaluates to TRUE, it writes this last event to register r_1, displayed as a dashed square in Figure 1. On the other hand, the transition from q_1 to q_f uses both the current element and the element stored in r_1 $((\sim, r_1))$ and passes them to the binary formula ϕ_2. The condition $\top$ (in the self-loop of q_1) is unary, always evaluates to TRUE and allows us to skip and ignore any number of events. The *SRT* of Figure 1 captures the *SREMO* of eq. (1). ◄

We can describe formally the rules for the behavior of a *SRT* through the notion of configuration:

Definition 4.3 (Configuration of SRT). Let $S = t_1, t_2, \cdots, t_l$ be a string and T a *SRT* consuming S. A configuration of T is a triple $c = [j, q, v]$, where

- j is the index of the next event/character to be consumed,
- q is the current state of T and
- v the valuation, i.e., the current contents of T's registers.

$c' = [j', q', v']$ is a *successor* of c iff one of the following holds:

- $\exists \delta : \delta.source = q$, $\delta.target = q'$, $\delta.\phi = \epsilon$, $j' = j$, $v' = v$, i.e., if this is an ϵ transition, we move to the target state without changing the index or the registers' contents.
- $\exists \delta : \delta.source = q$, $\delta.target = q'$, $\delta.W = \emptyset$, $(t_j, v) \models \delta.\phi$, $j' = j + 1$, $v' = v$, i.e., if the condition is satisfied according to the current event and the registers' contents and there are no write registers, we move to the target state, we increase the index by 1 and we leave the registers untouched.
- $\exists \delta : \delta.source = q$, $\delta.target = q'$, $\delta.W \neq \emptyset$,$(t_j, v) \models \delta.\phi$, $j' = j + 1$, $v' = v[W \leftarrow t_j]$, i.e., if the condition is satisfied according to the current event and the registers' contents and there are write registers, we move to the target state, we increase the index by 1 and we replace the contents of all write registers (all $r_i \in W$) with the current element from the string. ◄

We denote a succession of configurations by $[j, q, v] \rightarrow [j', q', v']$, or $[j, q, v] \xrightarrow{\delta} [j', q', v']$ if we need to refer to the transition as well. For the initial configuration, before any elements have been consumed, we assume that $j = 1$, $q = q_s$ and $v(r_i) = \sharp$, $\forall r_i \in R$. In order to move to a successor configuration, we need a transition whose condition evaluates to TRUE, when applied to $\sim$, if it is unary, or to $\sim$ and the contents of its register selection, if it is n-ary. If this is the case, we move one position ahead in the stream and update the contents of this transition's write registers, if any, with the event that was read. If the transition is an ϵ-transition, we do not move the stream pointer (since ϵ transitions are followed "spontaneously", without reading any events) and do not update the registers, but only move to the next state.

The actual behavior of a *SRT* upon reading a stream is captured by the notion of the run:

Definition 4.4 (Run of SRT over string/stream). A run ϱ of a *SRT* T over a stream $S = t_1, \cdots, t_n$ is a sequence of successor configurations $[1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} [n+1, q_{n+1}, v_{n+1}]$. A run is called accepting iff $q_{n+1} \in T.Q_f$ and $\delta_n.o = \bullet$. By $Match(\varrho)$ we denote all the indices in the string that were "marked" by the run, i.e., $Match(\varrho) = \{i \in [1, n] : \delta_i.o = \bullet\}$. ◄

The set of all runs over a stream S that T can follow is denoted by $Run(T, S)$ and the set of all accepting runs by $Run_f(T, S)$.

Example 4.5. An accepting run of the *SRT* of Figure 1, while consuming the first four events from the stream of Table 1, is the following:

$$\varrho = [1, q_s, \sharp] \xrightarrow{\delta_{s,1}} [2, q_1, (B, 1, 22, 300)] \xrightarrow{\delta_{1,1}} [3, q_1, (B, 1, 22, 300)] \xrightarrow{\delta_{1,1}} [4, q_1, (B, 1, 22, 300)] \xrightarrow{\delta_{1,f}} [5, q_f, (B, 1, 22, 300)] \tag{2}$$

Transition subscripts in this example refer to states of the *SRT*, e.g., $\delta_{s,1}$ is the transition from the start state to q_1, etc. Note that the valuation (contents or register r_1) changes only once, from $\sharp$ (empty) to $(B, 1, 22, 300)$, after the transition from q_s to q_1 with the first event. For the remaining configurations, the valuation remains the same. This is the only transition that writes to r_1. The contents of r_1 are retrieved and used in the last transition, from q_1 to q_f. Run (2) is not the only run, since the *SRT* could have followed other transitions with the same input, e.g., moving directly from q_s to q_1. Another possible (and non-accepting) run would be the one where the *SRT* always remains in q_1 after its first transition. ◄

Finally, we can define the language of a *SRT* as the set of strings for which the *SRT* has an accepting run, starting from an empty configuration. Similarly, the matches of a *SRT* on a string are the matches the *SRT* "produces" by marking the input elements as it reads the string, starting from an empty configuration.

Definition 4.6 (Language recognized and matches detected by SRT). We say that a *SRT* T accepts a string S iff there exists an accepting run $\varrho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} [n+1, q_{n+1}, v_{n+1}]$ of T over S, where $q_1 = T.q_s$ and $v_1 = \sharp$. The set of all strings accepted by T is called the language recognized by T and is denoted by $Lang(T)$. The set of matches detected by T on a string S is defined as $Match(T, S) = \{Match(\varrho) \mid \varrho \in Run_f(T, S)\}$. ◄

We now study the properties of *SRT*. All proofs are presented in the extended technical report. We first prove that, for every *SREMO* there exists an equivalent *SRT*. The proof is constructive, similar to that for classical automata. Equivalence between an expression e and a *SRT* T means that they recognize the same language and have the same matches. See Definitions 3.6 and 4.6.

Theorem 4.7. *For every SREMO e there exists an equivalent SRT T, i.e., a SRT such that $Lang(e) = Lang(T)$ and $Match(e, S) = Match(T, S)$ for every string S.*

Next, we study the closure properties of *SRT*. Informally, closure under some operator (e.g., union) means that we can always begin with some initial set of *SRT* and use them to construct a new *SRT*, such that its language (matches) will be the result of applying the operator on the languages (matches) of the initial set of *SRT*. For example, closure under union means that, for every pair of *SRT* T_1 and T_2, there exists a *SRT* T such that M is a match of T iff it is a match of T_1 or T_2. The formal definitions for closure under various operators may be found in the technical report.

Theorem 4.8. *SRT are closed under union, concatenation and Kleene-star.*

Example 4.9. The *SRT* of Figure 1 is equivalent to *SREMO* (1). ◄

SRT can thus be constructed from the three basic operators in a compositional manner, providing substantial flexibility and expressive power for CER applications.

Thus far we have described a basic set of operators with which we can define complex event patterns and their corresponding computational model. We have shown that our framework, with these basic operators, has unambiguous, compositional semantics. Contrary to previous CER systems, it does not impose ad hoc restrictions on the use of the operators, which may be used in a fully compositional manner. Besides concatenation/sequence, union/disjunction and Kleene-star/iteration, CER systems make extensive use of other operators as well and even constructs which are external to the language itself: the operators of intersection/conjunction and complement/negation, the use of deterministic automata for CER, the use of windows and of selection strategies.

We first study the closure properties of *SRT* under intersection and complement, two popular operators in CER. We can prove the following (see technical report for the proof):

Theorem 4.10. *SRT are closed under intersection but not under complement.*

Intersection was not defined as an operator of *SREMO* in Definition 3.4. Theorem 4.10 shows that we can introduce such an operator. On the other hand, as is the case for register automata [30], *SRT* are not closed under complement. This result could pose difficulties for handling *negation*, i.e., the ability to state that a sub-pattern should not happen for the whole pattern to be detected. However, we can (partially) overcome this problem by using windows in *SREMO* and *SRT*, i.e., by limiting the length of strings accepted by *SREMO* and *SRT*. In general, CER systems are not expected to remember every past event of a stream and produce matches involving events that are very distant. On the contrary, it is usually the case that CER patterns include an operator that limits the search space of input events, through the notion of windowing.

For a windowed *SREMO*, it is possible to construct an equivalent deterministic *SRT*, provided that we ignore the outputs of this *SRT*, essentially treating it as a recognizer and not as a transducer. Therefore, equivalence is proven by showing that the language of a windowed *SREMO* and that of a *SRT* are the same, ignoring equivalence based on the produced matches. We call such automata output-agnostic *SRT*.

Theorem 4.11. *For every windowed SREMO there exists an equivalent output-agnostic deterministic SRT.*

Corollary 4.12. *Output-agnostic SRT compiled from windowed SREMO are closed under complement.*

This result allows us to extend windowed *SREMO* to also include a negation operator. Although the result about closure under complement holds only when outputs are ignored, this is a minor limitation since we are not typically interested to mark elements that are negated in a *SREMO*.

CER patterns are sometimes characterized by their so-called *selection strategy* [25]. This strategy determines whether the input events in a match should occur contiguously in a stream or intermittently, with other, irrelevant events happening between the relevant ones. strict-contiguity, skip-till-any-match and skip-till-next-match are the three common such strategies. We can show that selection strategies may be applied as operators, through certain rewriting rules. This implies that multiple (even nested) strategies may be used in a pattern (more details may be found in the technical report).

For example, skip-till-any-match may be defined as follows:

Definition 4.13 (skip-till-any-match). If $e_1, e_2, \cdots, e_n$ are *SREMO*, then $e_{any} := \circlearrowleft (e_1, e_2, \cdots, e_n)$ is a *SREMO* with $\circlearrowleft$ denoting the skip-till-any-match selection strategy and

$$e_{any} := e_1 \cdot (\top \uparrow \otimes)^* \cdot e_2 \cdot (\top \uparrow \otimes)^* \cdots (\top \uparrow \otimes)^* \cdot e_n$$

In summary, we can state the following. Intersection is an operator that can be supported by our framework without any constraints. Negation and determinization can also be supported, but only for windowed expressions and with the understanding that negated events cannot be marked as being part of a match. With respect to selection strategies, skip-till-any-match can be accommodated without any constraints. skip-till-next-match is also available, with some minor limitations.

5 EXPERIMENTAL RESULTS

We have implemented a *SRT*-based CER engine by extending Wayeb[2]. We present our implementation and experimental results.

5.1 Implementation

The workflow of our engine is the following (see Algorithm 1). The user provides a pattern in the form of a windowed *SREMO* with a specific selection strategy and the engine compiles this pattern into a *SRT* T_s. This *SRT* is then fed with a stream S of simple events. Initially, before any input event has been consumed, the set of runs $Run(T_s, S_{..0})$ is composed of a single run (see Definition 4.4), $[1, T'.q_s, \sharp]$. $S_{..0}$ denotes the stream when no event has yet been processed. The single run, $[1, T'.q_s, \sharp]$, points to the first event in the stream, it is in its start state q_s and its registers are empty ($\sharp$). Wayeb then reads input events one by one and updates its set of runs after every new event. At each timepoint k, before reading the k^{th} event t_k, Wayeb maintains the set $Run(T_s, S_{..k-1})$. After processing t_k, it produces $Run(T_s, S_{..k})$. This is achieved by evaluating t_k against every $\varrho \in Run(T_s, S_{..k-1})$. Each run $\varrho = [1, q_1, v_1] \overset{\delta_1}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k, v_k]$ has to evaluate t_k on all the outgoing transitions of state q_k. If no transition is triggered, this means that the *SRT* cannot move to another state and ϱ is thus discarded and not included in $Run(T_s, S_{..k})$. If only one transition is triggered, then ϱ is updated, becoming $\varrho = [1, q_1, v_1] \overset{\delta_1}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k, v_k] \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}, v_{k+1}]$, with a new state q_{k+1} and register contents v_{k+1}. If n transitions are triggered and thus n next states are to be reached, then ϱ is replaced by n new runs ϱ', ϱ'', etc. Then each of these runs is updated with the new state and register contents

- $\varrho' = [1, q_1, v_1] \overset{\delta_1}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k, v_k] \overset{\delta'_k}{\rightarrow} [k+1, q'_{k+1}, v'_{k+1}]$
- $\varrho'' = [1, q_1, v_1] \overset{\delta_1}{\rightarrow} \cdots \overset{\delta_{k-1}}{\rightarrow} [k, q_k, v_k] \overset{\delta''_k}{\rightarrow} [k+1, q''_{k+1}, v''_{k+1}]$
- ...

The updated/new runs are added to the set of runs $Run(T_s, S_{..k})$, replacing ϱ. Accepting runs are treated specially. If $q_{k+1} \in T_s.Q_f$ and $\delta_k.o = \bullet$ for some run ϱ, then ϱ reports all the input events that it has marked with $\bullet$ and is then "killed", i.e., not added to $Run(T_s, S_{..k})$. This process is repeated for all runs of $Run(T_s, S_{..k-1})$.

Algorithm 1: Running Wayeb with non-deterministic *SRT*.

```
Input: SRT $T_s$, input event $t_k$, active runs $Run(T_s, S_{..k-1})$
Output: Active runs $Run(T_s, S_{..k})$, accepting runs
        $Run_f(T_s, S_{..k})$
$Run_f(T_s, S_{..k}) \leftarrow \emptyset$;
$Run(T_s, S_{..k}) \leftarrow \emptyset$;
foreach $\varrho \in Run(T_s, S_{..k-1})$ do
    $C \leftarrow FindSuccessorConfigurations(\varrho, t_k)$;
    if $|C| > 0$ then
        $c \leftarrow$ pick and remove element from $C$;
        $\varrho_{new} \leftarrow UpdateRun(\varrho, c)$;
        if $IsAccepting(\varrho_{new})$ then
            $ReportMatch(\varrho_{new})$;
            $Run_f(T_s, S_{..k}) \leftarrow Run_f(T_s, S_{..k}) \cup \varrho_{new}$;
        else
            $Run(T_s, S_{..k}) \leftarrow Run(T_s, S_{..k}) \cup \varrho_{new}$;
        foreach $c \in C$ do
            $\varrho' \leftarrow Clone(\varrho)$;
            $\varrho_{new} \leftarrow UpdateRun(\varrho', c)$;
            if $IsAccepting(\varrho_{new})$ then
                $ReportMatch(\varrho_{new})$;
                $Run_f(T_s, S_{..k}) \leftarrow Run_f(T_s, S_{..k}) \cup \varrho_{new}$;
            else
                $Run(T_s, S_{..k}) \leftarrow Run(T_s, S_{..k}) \cup \varrho_{new}$;
return $Run_f(T_s, S_{..k})$,$Run(T_s, S_{..k})$;
```

The cost of evaluating a single event t_k depends on several factors. It depends on $|Run(T_s, S_{..k-1})|$, the number of active runs against which t_k is to be evaluated. It also depends on the number of outgoing transitions from the states of active runs as well as on the complexity of evaluating the predicates of transitions. If we

[2] https://github.com/ElAlev/Wayeb

assume a constant cost for predicate evaluation c_p and then bound the number of outgoing transitions to be at most n_p, where n_p is the number of predicates appearing in the initial *SREMO* (including the $\top$ predicate), then the cost of evaluating t_k against a run ϱ is at most $n_p \cdot c_p$. Therefore, the total cost of evaluating all runs is $|Run(T_s, S_{..k-1})| \cdot n_p \cdot c_p$. In the worst case, all outgoing transitions of all runs are triggered. We will thus have to create $|Run(T_s, S_{..k-1})| \cdot (n_p - 1)$ new run clones and perform $|Run(T_s, S_{..k-1})| \cdot n_p$ run updates. If c_c is the cost of run cloning and c_u the cost of run updating, then the total cost would be

$$\begin{aligned} c =& |Run(T_s, S_{..k-1})| \cdot n_p \cdot c_p + |Run(T_s, S_{..k-1})| \cdot (n_p - 1) \cdot c_c + \\ & |Run(T_s, S_{..k-1})| \cdot n_p \cdot c_u && = \\ & |Run(T_s, S_{..k-1})| \cdot (n_p \cdot c_p + \cdot (n_p - 1) \cdot c_c + n_p \cdot c_u) && = \\ & |Run(T_s, S_{..k-1})| \cdot (n_p \cdot (c_p + c_c + c_u) - c_c) \end{aligned} \tag{3}$$

The complexity depends highly on the number of active runs at every timepoint. We can also estimate the runtime complexity on a "per-window" basis, by attempting to calculate the total number of runs created for a window of input events. Relevant results have been obtained in [45]. For a sequential pattern (without disjunction or Kleene-star) under strict-contiguity and a window w, the total number of created runs is $R \cdot w$, where R is the percentage of input events satisfying predicate p of the outgoing transition from the a state. Under strict-contiguity, there is only one state where cloning may occur and this is the first state, which has a self-loop with $\top$ and a transition to another state with predicate p. This predicate will be satisfied $R \cdot w$ times. If the average cost of handling a run is c_r (including predicate evaluation, clone creation, etc.), then the total cost is $R \cdot w \cdot c_r$. Under skip-till-any-match, the first state will create $R \cdot w$ clones, the second $(R \cdot w)^2$, etc. We thus have a geometric series and the total number of created runs will be $\frac{(R\cdot w)^{i+1}-1}{(R\cdot w)-1}$, where i is the number of "terminal" sub-patterns in the original pattern. If the pattern contains j Kleene "components" (and thus the automaton j states with self-loops), then the total number of runs will be $\frac{(R\cdot w)^{i-j+1}-1}{(R\cdot w)-1} \cdot 2^{j\cdot R\cdot w}$. We see then that the worst-case cost becomes exponential in the size of the window and the number of Kleene-star operators.

5.2 Experimental setup

We present experimental results by comparing Wayeb against other state-of-the-art CER systems. Our goal is to test the systems with expressive, relational patterns, i.e., with patterns which can relate multiple events. For this reason, we had to exclude systems that cannot express relational patterns, such as CORE and previous versions of Wayeb. For some other systems, there is no publicly available implementation or the implementation is no longer maintained (e.g., CRS and Cayuga). Yet some other systems (e.g., TESLA) suffer from low performance for certain classes of queries [12].

Flink's implementation of MATCH_RECOGNIZE [3] was also considered. However, though rich with various features, it is limited in certain crucial respects. For example, iteration can only be applied to single events and not to subsequences. Moreover, we were not able to reproduce results obtained from other engines, even with simple sequential patterns, when applying the skip-till-any-match strategy. Several matches were missing from the output. Nevertheless, we attempted to run some experiments and measure Flink's throughput, even when it failed to report all matches. We discovered that its throughput was the lowest of all other engines and comparable to that of FlinkCEP, Flink's CER engine. This is not a surprising result, as Flink's implementation of MATCH_RECOGNIZE is based on FlinkCEP. For these reasons, we excluded MATCH_RECOGNIZE from any further experiments.

Our comparison thus includes SASE v1.0 [7], Esper v8.7.0 [1] and FlinkCEP v1.16.1 [4]. All these engines are written in Java. Wayeb is implemented in Scala 2.12.10. All experiments were run on a 64-bit Linux machine with AMD EPYC 7543 × 126 processors and 400 GB of memory. We used Java 1.8 for all systems. All experiments for all systems were run as single-core applications. Wayeb is an open-source engine and the presented experiments are reproducible[3].

As a basis for our experiments, we used the benchmark suite presented in [13][4]. The suite contains three datasets: a) stock market data from a single day (224,473 input events); b) plug measurements from smart homes (1,000,000 input events) and c) taxi trips from the city of New York (585,762 input events). For the stock market dataset, each input event is a BUY or SELL event, containing the name of the company, the price of the stock, the volume of the transaction and its timestamp. For the smart homes dataset, each input event is a LOAD event, containing a load value in Watts, a household id, a plug id and a timestamp. For the taxis dataset, each input event is a TRIP event, containing the datetime of the pickup and dropoff, the zone of the pickup and dropoff, the trip distance and duration, the fare amount, the tip amount, etc.

Our results are presented incrementally as we increase the complexity of the tested patterns. We start with sequential patterns where some of the simple events are related through constraints (Section 5.3). We also study the effect of window size on such patterns (Section 5.4). We then add Kleene operators on single events to these patterns (Section 5.5). We additionally test patterns with nested Kleene operators (Section 5.6). Finally, we present results with patterns containing various, mixed operators (Section 5.7). Since windows are ubiquitous in CER (for performance issues), we decided to focus on windowed *SREMO* in our experiments. We also fixed the selection strategy to skip-till-any-match, since this is the most demanding strategy, both in terms of time and space complexity. For all experiments described here, we have made sure that all engines produce the same results for each pattern.

The benchmark suite runs each experiment, i.e., each combination of engine, pattern and window size, 3 times. We report the average throughput and memory footprint. Throughput is measured in terms of (input) events processed per second, whereas memory is measured in terms of used memory (MB). For each run, multiple memory measurements are taken, one every 10.000 input events. Before the measurement, the garbage collector is explicitly called. We report the average of those memory measurements. The time we use to calculate throughput includes both the time required to process input events (update the state(s) of the automaton, create new runs, discard old ones, etc.) and the time required to report any complex events. However, we have slightly modified the notion of

[3] https://github.com/ElAlev/cer-srt.

[4] https://github.com/CORE-cer/CORE-experiments.

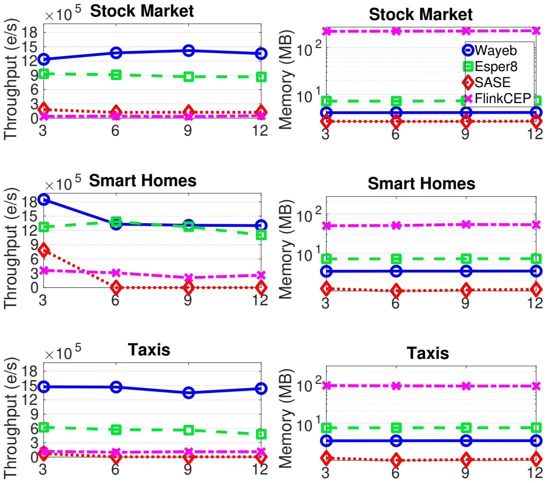

Figure 2: Throughput and memory consumption for sequential patterns with n-ary predicates as a function of pattern length. Window sizes are $w_{stock} = 500$, $w_{smart} = 5$, $w_{taxi} = 100$.

"reporting a complex event". Instead of writing it in a file/database (a system-dependent, expensive operation), we perform (for all systems) a simple arithmetic operation on the timestamps of its constituent simple events.

We considered using implementation-independent metrics to compare the different systems. However, the different implementations vary widely and do not necessarily share common operators which could act as basic measurement blocks. This is especially true for Esper, which, besides automata, also employs trees and Allen's interval algebra. For this reason, we decided to follow previous work on comparing different CER systems, where throughput is used as a metric [13, 45]. Note, however, that the compared systems are all JVM-based, thus significantly limiting the effect of language choice on their performance. With respect to complexity, the publicly available implementation of SASE is very similar to Wayeb. Thus, they have similar complexities. However, their performance might vary significantly due to differences in the constants of Eq. (3) concerning the costs of run cloning/updating. Concerning FlinkCEP, according to its source code [5], it closely follows the version of SASE presented in [8]. It is not clear which optimizations are actually implemented and what their effects on FlinkCEP's complexity are. Finally, Esper's documentation discusses the complexity of some operations, but not those of pattern matching [2].

5.3 Sequential patterns

Our first set of experiments is focused on sequential patterns. We begin with patterns of the following form:

$$seq_3 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \quad (4)$$

where $\circlearrowleft$ denotes the skip-till-any-match selection strategy (see Definition 4.13), w is the window size and ϕ_i all contain "local" constraints, i.e., conditions applied to the single, most recently read event, while ϕ_3 relates the most recently read input event with the event that triggered ϕ_1. For example, in the stock market dataset, we have:

$$\phi_1(x) := x.name = INTC$$
$$\phi_2(x) := x.name = RIMM$$
$$\phi_3(x, y) := (x.name = QQQ \land x.price > y.price)$$

This specific pattern captures a sequence of three stock ticks from three given companies. The relational constraint is that the stock price of the last event should be greater than the price of the first event. For each such pattern, we run experiments for variable pattern "length". We say that the length of the Pattern in eq. (4) is 3 because it is composed of 3 terminal sub-expressions. We can increase the length of the pattern by adding more such expressions. In our experiments we have used patterns of length 3, 6, 9 and 12. For example, the pattern of length 6 has the following form:

$$\begin{aligned} seq_6 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim, r_1) \uparrow \bullet), \\ (\phi_4(\sim) \uparrow \bullet \downarrow r_2), (\phi_5(\sim) \uparrow \bullet), (\phi_6(\sim, r_2) \uparrow \bullet))^{[1..w]} \end{aligned} \quad (5)$$

The general template remains the same, i.e., ϕ_4, ϕ_5 and ϕ_6 all apply local filters with a given company name. For every three new subexpressions we also add a relational constraint (e.g., between ϕ_4 and ϕ_6 in Pattern (5)). The window size is kept constant (e.g., for the stock market dataset, $w = 500$). The match frequency (ratio of complex to input events) is in the range of $0.36\% - 0\%$ for the stock market dataset (the lengthier the pattern the lower the number of detected matches), $0.36\% - 0\%$ for the smart homes dataset and $0.05\% - 0\%$ for the taxis dataset. Note that our purpose in using such patterns is to stress test the systems under controlled conditions. Some of the patterns may not be intuitive from a practical point of view, but allow for controlled experiments. This is the reason why we use a symmetrical and repeatable structure in the patterns when increasing their length. We aim at testing the effect of length, without introducing other performance affecting factors.

Figure 2 presents throughput and memory results for the aforementioned sequential patterns and for all datasets. Wayeb and Esper stand out clearly as the most efficient engines in terms of throughput. Wayeb also has a significant advantage over Esper in most experiments and a slight advantage for the smart homes dataset. For example, Wayeb is almost 2.5 times as efficient as Esper for the taxis dataset. FlinkCEP has by far the heaviest memory footprint, while the other systems seem to have a similar performance. Wayeb has a slightly better performance than Esper, its main competitor in terms of throughput. In general, we see that the performance is relatively stable as a function of pattern length for all systems. This is especially true for memory. In particular, SASE's low memory footprint can be attributed to its general lightweight construction (the other systems are designed to perform additional tasks, besides vanilla, single-core CER) and its memory optimization schemes, such as run recycling. Throughput exhibits slight variations. This observation implies that the number of created runs does not vary greatly for the tested sequential patterns.

5.4 Varying window size

In the next set of experiments, we investigated the behavior of all systems with increasing window sizes. For each dataset, we increased the window size up to the point where throughput exhibits a significant drop. Figure 3 shows the relevant results. Wayeb again

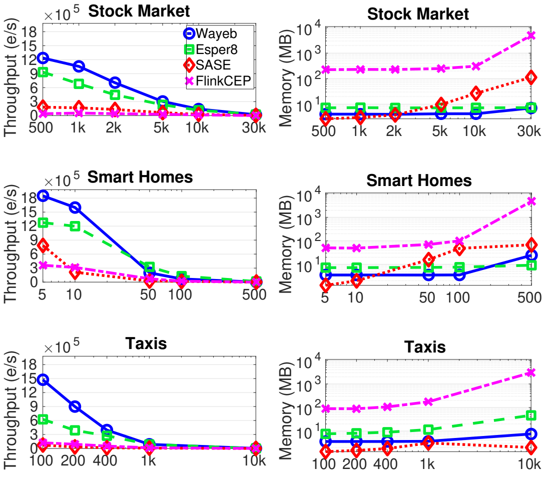

Figure 3: Throughput and memory consumption for sequential patterns with *n*-ary predicates as a function of window size. Pattern length is 3.

exhibits the best performance in terms of throughput, followed by Esper. Moreover, Wayeb remains better than Esper and FlinkCEP in terms of memory consumption. All systems exhibit a throughput deterioration as the window size increases. This implies that window size is more important in determining the number of created runs than pattern length. Wayeb and Esper also show a stable memory footprint, indicating that the memory space reserved for the number of runs is small compared to the total space required by the engines. This conclusion is reinforced by SASE's memory deterioration. As a bare-bones CER engine, its memory consumption is dominated by the number of runs, which is visible in the results.

5.5 Patterns with Kleene operators

We now move to patterns containing Kleene operators. We tested the engines against patterns of the following form:

$$k_3 := \circlearrowleft\, ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet)^+, (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \quad (6)$$

Pattern (6) is the same as Pattern (4), with a single difference. The middle sub-expression (ϕ_2) may be repeated more than once, by using a Kleene-plus operator ($\phi^+ := \phi \cdot \phi^*$). Again, we use patterns of length 3, 6, 9, 12, gradually increasing the number of Kleene operators (e.g., patterns of length 6 have 2 such operators). The match frequencies are in the range of $0.61\% - 0\%$, $1.35\% - 0\%$, $0.08\% - 0\%$ for the stock market, smart homes and taxis dataset.

Figure 4 shows the throughput results (left column). We excluded SASE and FlinkCEP from this set of experiments because they cannot support patterns with Kleene operators with the expected semantics. As far as SASE is concerned, although it can accept, compile and run patterns with Kleene operators, it tends to produce many more matches than those expected from the semantics of skip-till-any-match. This indicates that SASE could possibly suffer from soundness issues, at least when some operators are used. FlinkCEP, on the other hand, has the inverse problem. Our investigation of FlinkCEP has led us to conclude that this behavior is probably due to the fact that FlinkCEP does not allow the use of skip-till-any-match within a Kleene operator. Some matches are thus dropped. For these reasons, we focused on Wayeb and Esper which can support patterns with Kleene operators and skip-till-any-match.

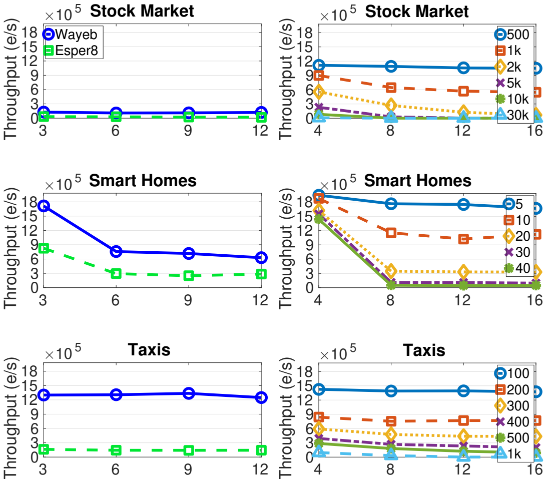

Figure 4: Throughput for patterns with *n*-ary predicates and Kleene operators as a function of pattern length (left). Throughput for patterns with *n*-ary predicates and nested Kleene operators as a function of pattern length for various windows (right).

Wayeb always exhibits higher throughput than Esper. In some cases (e.g., for the taxis dataset), Wayeb's throughput is 6 times that of Esper's. As expected, the performance of both Wayeb and Esper for this class of patterns is lower than their performance for sequential patterns. Due to the presence of Kleene operators, the engines need to produce many more runs. Whenever the stream contains simple events satisfying ϕ_2, the engines need to keep track of all possible combinations of these events. This is the reason why more runs are created. Both Wayeb and Esper have a stable memory usage across all patterns: less than 5 and 10 MB respectively.

5.6 Patterns with nested Kleene operators

At the next level of pattern complexity, we have patterns with nested Kleene operators. In order to run experiments with such patterns, we used expressions of the following form:

$$\begin{aligned} kn_4 := \circlearrowleft\, ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), ((\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim) \uparrow \bullet)^+)^+, \\ (\phi_4(\sim, r_1) \uparrow \bullet))^{[1..w]} \end{aligned} \quad (7)$$

Note that ϕ_2 is under a single Kleene-plus operators whereas ϕ_3 under two. This expression has 4 terminal sub-expressions. We also used patterns with 8, 12 and 16 terminal sub-expressions, i.e., patterns with multiple (2, 3 and 4) nested Kleene operators. The match frequencies are in the range of $4.29\% - 2.000\%$, $0.08\% - 0.64\%$, $0.03\% - 1.19\%$ for the stock market, smart homes and taxis dataset. These numbers correspond to the extreme window values of the shortest query kn_4 (the larger the window the more the detected matches). For lengthier queries, the frequency may drop to 0.0%.

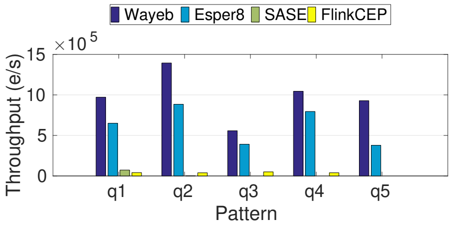

Figure 5: Throughput for patterns with n-ary predicates and various operators. $w = 1000$.

SASE's language does not support patterns with nested Kleene operators. FlinkCEP has the issues mentioned in Section 5.5 regarding the semantics of skip-till-any-match with iteration. Esper's language is also not able to support such patterns. Wayeb is the only engine which can properly support nested Kleene operators.

The experimental results are shown in Figure 4 (right column). In order to gain a more complete understanding of Wayeb's behavior, we show results for multiple values of the window size. Wayeb maintains high throughput, in the order of millions or hundreds of thousands of events per second, for most combinations of window size and pattern length. As the window size increases, Wayeb's performance deteriorates, since larger window sizes always lead to more runs being created. The combined variation of window size and pattern length in this figure illustrates also the more pronounced combined effect on throughput. The window size still remains the most important factor for performance. For large windows though, the pattern length starts having an impact as well, since larger windows give a chance to longer patterns to create additional runs.

5.7 Patterns with other operators

In the last set of experiments, we used the stock market dataset and tested all engines against patterns with various operators. We considered a diverse range of patterns, where other operators like disjunction, iteration and their combination were employed. In particular, we tested 5 patterns:

(1) A sequential pattern starting and ending with a SELL event, and with two BUY events in between.

$$q_1 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim) \uparrow \bullet), (\phi_4(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{8}$$

where

$$\phi_1(x) := x.type = SELL \land x.name = MSFT$$
$$\phi_2(x) := x.type = BUY \land x.name = ORCL$$
$$\phi_3(x) := x.type = BUY \land x.name = CSCO$$
$$\phi_3(x, y) := x.type = SELL \land x.name = AMAT \land x.price < y.price$$

(2) q_2: same as q_1, but with local thresholds on price.

(3) q_3: same as q_1, but ϕ_2 now includes disjunction: $\phi_2(x) := (x.type = BUY \lor x.type = SELL) \land x.name = ORCL$. We also applied the same modification to ϕ_3.

(4) q_4: same as q_3, but with local thresholds on price.

(5) Combining iteration and disjunction:

$$q_5 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet)^+, (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{9}$$

where

$$\phi_2(x) := (x.type = BUY \lor x.type = SELL) \land x.name = QQQ \land x.volume = 4000$$

SASE can only support *SREMO* q_1 and q_2. Therefore, we do not show SASE results for *SREMO* q_3, q_4 and q_5. FlinkCEP supports all 5 patterns, but its semantics of the iteration operator are ambiguous and its results when using iteration do not match those of the other systems. Therefore, we do not show FlinkCEP results for *SREMO* q_5. The match frequencies are in the range of 0% (q_2) to 13% (q_3).

The relevant results are shown in Figure 5. Wayeb has the highest throughput for all patterns, followed by Esper. The performance for q_2 is higher than that for q_1, due to the presence of extra threshold filters which prune several runs. On the other hand, q_3 is the most demanding one, because it does not have any threshold filters and it includes disjunction, thus leading to more runs being created. q_4 rebounds to higher throughput figures, due to the inclusion of filters. For q_5, Esper has its lowest performance and Wayeb its second lowest, due to the presence of both iteration and disjunction.

Finally, we experimentally tested Wayeb's performance on the above patterns when there is no requirement for it to produce an output, i.e., to completely enumerate each complex event. For this purpose, we completely switched off Wayeb's functionality of gradually creating partial matches. We only retained its functionality of tracking the runs to determine whether they have reached a final state. Wayeb's performance remained almost unaffected. The reason for this behavior is that we already represent runs in a very minimal way, even when they need to carry partial matches. Thus, the main bottleneck for Wayeb lies in the actual evaluation and maintenance of the runs and not in the production of their output.

6 SUMMARY & FUTURE WORK

We presented a system for CER based on an automaton model, *SRT*, that supports patterns with n-ary conditions ($n \geq 1$), which are quintessential for CER applications. *SRT* have nice compositional properties, as most of the standard operators in CER, such as concatenation / sequence, union / disjunction, intersection / conjunction and Kleene-star / iteration, may be used freely. Complement may also be used and determinization is possible, if a window operator is used, a very common feature in CER. The experimental results show that our framework with *SRT* is highly expressive, with the ability to support complex patterns with nested operators and relational constraints, while outperforming other engines for most patterns and workloads.

Our aim for the future is to investigate our engine's optimization potential, given that its current implementation is straightforward. We will also explore the ability of *SREMO* and *SRT* to capture aggregates, which is related to how the current semantics of *SREMO* determine (register) variable binding. We intend to explore how *SREMO* can efficiently capture aggregates. Finally, we will investigate how we can extend *SRT* so that they can handle hierarchies of events, through the use of complete histories of events [44].

ACKNOWLEDGMENTS

Supported by the CREXDATA project (EU Horizon2020 No 101092749).

REFERENCES

[1] [n.d.]. Esper. https://www.espertech.com/esper/. [Online; accessed 23-May-2024].

[2] [n.d.]. Esper complexity. http://esper.espertech.com/release-8.9.0/reference-esper/html/performance.html. [Online; accessed 23-May-2024].

[3] [n.d.]. Flink - Pattern Recognition. https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/table/sql/queries/match_recognize/. [Online; accessed 23-May-2024].

[4] [n.d.]. FlinkCEP - Complex event processing for Flink. https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/libs/cep/. [Online; accessed 23-May-2024].

[5] [n.d.]. FlinkCEP NFA source code. https://github.com/apache/flink/blob/master/flink-libraries/flink-cep/src/main/java/org/apache/flink/cep/nfa/NFA.java. [Online; accessed 23-May-2024].

[6] [n.d.]. ISO/IEC 19075-5:2021 Information technology — Guidance for the use of database language SQL — Part 5: Row pattern recognition. https://standards.iteh.ai/catalog/standards/iso/f753ca23-4b3c-4c9f-8a0a-1113f39bc404/iso-iec-19075-5-2021. [Online; accessed 23-May-2024].

[7] [n.d.]. SASE Open Source System. https://github.com/haopeng/sase. [Online; accessed 23-May-2024].

[8] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *SIGMOD Conference*. ACM, 147–160.

[9] Elias Alevizos, Alexander Artikis, and George Paliouras. 2018. Wayeb: a Tool for Complex Event Forecasting. In *LPAR (EPiC Series in Computing)*, Vol. 57. EasyChair, 26–35.

[10] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. 2022. Complex event forecasting with prediction suffix trees. *VLDB J.* 31, 1 (2022), 157–180.

[11] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-variable logic on data words. *ACM Trans. Comput. Log.* 12, 4 (2011), 27:1–27:26.

[12] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. 2021. CORE: a COmplex event Recognition Engine. *CoRR* abs/2111.04635 (2021).

[13] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. 2022. CORE: a COmplex event Recognition Engine. *Proc. VLDB Endow.* 15, 9 (2022), 1951–1964.

[14] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *Proc. VLDB Endow.* 3, 1 (2010), 220–231.

[15] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *DEBS*. ACM, 50–61.

[16] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62.

[17] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. 2019. Symbolic Register Automata. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 11561. Springer, 3–21.

[18] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 47–67.

[19] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2005. *A general algebra and implementation for monitoring event streams*. Technical Report. Cornell University.

[20] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. 2006. Towards Expressive Publish/Subscribe Systems. In *EDBT (Lecture Notes in Computer Science)*, Vol. 3896. Springer, 627–644.

[21] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *CIDR*. www.cidrdb.org, 412–422.

[22] Christophe Dousson and Pierre Le Maigat. 2007. Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization. In *IJCAI*. 324–329.

[23] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action*. Manning Publications Company.

[24] Malik Ghallab. 1996. On Chronicles: Representation, On-line Recognition and Learning. In *KR*. Morgan Kaufmann, 597–606.

[25] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352.

[26] Alejandro Grez, Cristian Riveros, and Martín Ugarte. 2019. A Formal Framework for Complex Event Processing. In *ICDT (LIPIcs)*, Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:18.

[27] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2020. On the Expressiveness of Languages for Complex Event Recognition. In *ICDT (LIPIcs)*, Vol. 155. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:17.

[28] Sylvain Hallé. 2017. From Complex Event Processing to Simple Event Processing. *CoRR* abs/1702.08051 (2017). http://arxiv.org/abs/1702.08051

[29] Ulrich Hedtstück. 2017. *Complex event processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer Vieweg, Berlin.

[30] Michael Kaminski and Nissim Francez. 1994. Finite-Memory Automata. *Theor. Comput. Sci.* 134, 2 (1994), 329–363.

[31] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In *SIGMOD Conference*. ACM, 1023–1036.

[32] Leonid Libkin, Tony Tan, and Domagoj Vrgoc. 2015. Regular expressions for data words. *J. Comput. Syst. Sci.* 81, 7 (2015), 1278–1297.

[33] Leonid Libkin and Domagoj Vrgoc. 2012. Regular Expressions for Data Words. In *LPAR (Lecture Notes in Computer Science)*, Vol. 7180. Springer, 274–288.

[34] David C. Luckham. 2005. *The power of events - an introduction to complex event processing in distributed enterprise systems*. ACM.

[35] Periklis Mantenoglou, Dimitrios Kelesis, and Alexander Artikis. 2023. Complex Event Recognition with Allen Relations. In *KR*. 502–511.

[36] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*. ACM, 193–206.

[37] Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435.

[38] Dusan Petkovic. 2022. Specification of Row Pattern Recognition in the SQL Standard and its Implementations. *Datenbank-Spektrum* 22, 2 (2022), 163–174.

[39] Luc Segoufin. 2006. Automata and Logics for Words and Trees over an Infinite Alphabet. In *CSL (Lecture Notes in Computer Science)*, Vol. 4207. Springer, 41–57.

[40] Efthimis Tsilionis, Alexander Artikis, and Georgios Paliouras. 2022. Incremental Event Calculus for Run-Time Reasoning. *J. Artif. Intell. Res.* 73 (2022), 967–1023.

[41] Gertjan van Noord and Dale Gerdemann. 2001. Finite State Transducers with Predicates and Identities. *Grammars* 4, 3 (2001), 263–286.

[42] Margus Veanes. 2013. Applications of Symbolic Finite Automata. In *CIAA (Lecture Notes in Computer Science)*, Vol. 7982. Springer, 16–23.

[43] Margus Veanes, Nikolaj Bjørner, and Leonardo Mendonça de Moura. 2010. Symbolic Automata Constraint Solving. In *LPAR (Yogyakarta) (Lecture Notes in Computer Science)*, Vol. 6397. Springer, 640–654.

[44] Walker M. White, Mirek Riedewald, Johannes Gehrke, and Alan J. Demers. 2007. What is "next" in event processing?. In *PODS*. ACM, 263–272.

[45] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD Conference*. ACM, 217–228.

[46] Erkang Zhu, Silu Huang, and Surajit Chaudhuri. 2023. High-Performance Row Pattern Recognition Using Joins. *Proc. VLDB Endow.* 16, 5 (2023), 1181–1194.