Complex Event Recognition with Symbolic Register Transducers

Elias Alevizos
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
alevizos.elias@iit.demokritos.gr

Alexander Artikis
Department of Maritime Studies, University of Piraeus, Greece
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
a.artikis@unipi.gr

Georgios Paliouras
Institute of Informatics & Telecommunications, National Center for Scientific Research "Demokritos", Greece
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 study the closure properties of *SRT* under union, intersection, concatenation, Kleene closure, complement and determinization by extending previous relevant results from the field of languages and automata theory. We show that *SRT* are closed under various operators, but are not in general closed under complement and they are not determinizable. However, they are closed under these operations when a window operator, quintessential in Complex Event Recognition, is used. 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, and that allows for a systematic treatment of such automata. For *SRT* to work in pattern detection, we allow them to mark events from the input stream as belonging to a complex event or not, hence the name "transducers". We also present an implementation of *SRT* which can perform CER. We 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.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Pattern matching; Information systems → Data streaming

Keywords and phrases Finite Automata, Regular Expressions, Complex Event Processing, Symbolic Automata

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" [27, 38, 18]. Typically, an event has the structure of a tuple of values which might be numerical or

categorical. Time is of critical importance for CER and thus a temporal formalism is used in order to define the patterns to be detected. Such a pattern imposes temporal (and possibly atemporal) constraints on the input events, which, if satisfied, lead to the detection of a complex event. Atemporal constraints may be "local", applying only to the last event read from the input stream. For example, in streams from temperature sensors, the constraint that the temperature of the last event is higher than some constant threshold would constitute such a local constraint. More commonly, these constraints involve multiple events of the pattern, e.g., the constraint that the temperature of the last event is higher than that of the previous event. Complex events must often be detected with very low latency, which, in certain cases, may even be in the order of a few milliseconds [38, 25, 32].

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 (assuming the input events are temporally ordered). 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 [23, 8, 18]. They resemble both register automata [34], through their ability to store events, and symbolic automata [20], 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 [28] 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. A recently proposed formal framework for CER attempts to address these issues [28]. Its advantage is that it provides a logic for CER patterns, with denotational and compositional semantics, but without imposing severe restrictions on the use of operators. An automaton model is also proposed which may be conceived as a variation of symbolic transducers [20]. However, this automaton model can only handle "local" constraints, i.e., the formulas on their transitions are unary and thus are applied only to the last event read. A model which combines symbolic and register automata (called symbolic register automata) has recently been proposed in [19]. However, this work focuses on the more theoretical aspects of the proposed automaton model, without investigating how this model may be applied to CER (e.g., by providing a language appropriate for CER or by examining the effects of windows).

We propose a system for CER, based on an automaton model that is a combination of symbolic and register automata. It has the ability to store events and its transitions have guards in the form of n-ary conditions. These conditions may be applied both to the last event and to past events that have been stored. Conditions on multiple events are crucial in CER because they allow us to express many patterns of interest, e.g., an increasing trend in the speed of a vehicle. 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. They also extend the power of symbolic register automata, by allowing events in a stream to be marked as belonging to a pattern match or not. This feature is crucial in cases where we need to enumerate all complex events detected at any given timepoint (i.e., exactly report all simple events which compose the complex ones) instead of simply reporting that a complex event has been detected. 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 [36], 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 Symbolic Regular Expressions with Memory and Output (*SREMO*).
- We show how this framework subsumes, in terms of expressive power, previous similar attempts. It allows for nesting operators and selection strategies. It also allows n-ary expressions to be used as conditions in patterns, thus opening the way for 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 in a pattern. Constraints with multiple events are essential in CER, since they are required in order to capture many patterns of interest, e.g., an increasing or decreasing trend in stock prices. *SRT* also have the ability to mark exactly those simple events comprising a complex one.
- We study the closure properties of *SRT*. By extending previous results from automata theory, we show that, in the general case, *SRT* are closed under the most common operators (union, intersection, concatenation and Kleene-star), but not under complement and determinization. Failure of closure under complement implies that negation cannot be arbitrarily (i.e., in a compositional manner) used in CER patterns. The negative result about determinization implies that certain techniques (like forecasting) requiring deterministic automata are not applicable.
- We show that, by using windows, *SRT* are able to retain their nice closure properties, i.e., they remain closed under complement and determinization. Windows are an indispensable operator in CER because, among others, they limit the search space for pattern matching.
- We describe the implementation of 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.

▶ **Example 1.** We now introduce an example to provide intuition. The example is that of a set of stock market ticks. A stream is a sequence of input events, where each such event is a tuple of the form $(type, id, price, volume)$. The first attribute ($type$) is the type of transaction: S for SELL and B for BUY. The second one (id) is an integer identifier, unique for each company. It has a finite set of possible values. The third one ($price$) is a real-valued number for the price of a given stock. Finally, the fourth one ($volume$) is a natural number referring

Table 1 Example of a stream.

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	...

to the volume of the transaction. Table 1 shows an example of such a stream. We assume that events are temporally ordered and their order is implicitly provided through the index. We also assume that concurrent events cannot occur, i.e., each index is unique to a single event.

In Table 2 we have gathered the notation that we use throughout the paper, along with a brief description of every symbol.

2 Related Work

Due to their ability to naturally handle sequences of characters, automata have been extensively adopted in CER, where they are adapted in order to handle streams composed of tuples. Typical cases of CER systems that employ automata are the Chronicle Recognition System [26, 24], Cayuga [22, 23], TESLA [17], SASE [8, 49], CORE [28, 15] and Wayeb [12, 9]. There also exist systems that do not employ automata as their computational model, e.g., there are logic-based systems [44, 39] or systems that use trees [40], 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 [28, 30]. Specifically, in [28], 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 [27], and for a general review of CER systems, see [18]. 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.

2.1 Extended automaton models: theory

Outside the field of CER, research on automata has evolved towards various directions. Besides the well-known push-down automata that can store elements from a finite set to a stack, there have appeared other automaton models with memory, such as register automata, pebble automata and data automata [34, 41, 13]. For a review, see [43]. 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 [34]. At each transition, a register automaton may choose to

Table 2 Notation used throughout the paper.

Symbol	Meaning
$\mathcal{V}$, $\mathcal{U}$	vocabulary, universe
$\mathcal{L}$ ($\mathcal{L} \subseteq \mathcal{U}^*$)	a language over $\mathcal{U}$
$t_i \in \mathcal{U}$	term / character
$S = t_1, t_2, \cdots$, $S_{i..j} = t_i, \cdots, t_j$	stream / stream "slice" from index i to j
$f(t_1, \cdots, t_m)$	function
P, $\top$	relation, unary TRUE relation
ϕ	formula
$\mathcal{M}$	$\mathcal{V}$-structure
$\mathcal{M} \models \phi$	$\mathcal{M}$ models ϕ
$R = \{r_1, \cdots, r_k\}$	register variables
$v : R \hookrightarrow \mathcal{U}$	valuation
$F(r_1, \cdots, r_k)$	set of all valuations on R
$\sharp$, $\sim$	contents of empty register, automaton head
$(u, v) \models \phi$	condition ϕ satisfied by element u and valuation v
ϵ	the "empty" symbol
$\bullet$, $\otimes$	outputs
$e_1 + e_2$, $e_1 \cdot e_2$, e^*, $!e$	regular disjunction / concatenation / iteration / negation
$\circlearrowleft e$, $@e$	skip-till-any-match, skip-till-next-match operators
$e^{[1..w]}$	windowed expression with window size w
$(e, S, M, v) \vdash v'$	string S and match M on expression e with initial valuation v induce valuation v'
$Lang(e)$	language accepted by expression e
$Match(e, S)$	matches detected by e on S
T	automaton / transducer
Q, q^s, Q^f	automaton states / start state / final states
Δ, δ	automaton transition function / transition
W	write registers of a transition
$c = [j, q, v]$	automaton configuration (j current position, q current state, v current valuation)
$[j, q, v] \xrightarrow{\delta} [j', q', v']$	configuration succession
$\varrho = [1, q_1, v_1] \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}, v_{k+1}]$	run of automaton T over stream $S_{1..k}$
$Lang(T)$	language accepted by automaton T
$Match(T, S)$	matches detected by T on S

System	σ_1	σ_n	$\vee$	$\wedge$	$\neg$	;	*	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 $\neg$ 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	$\neg$ 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, $\neg$: negation, ;: sequence, *: iteration, D: determinizability, E: enumeration, S.P.: selection policies, Stam : skip-till-any-match, Stnm : skip-till-next-match, Sc : strict-contiguity.

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 [36, 37]. We build on the framework presented in [36, 37] in order 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 attributes of events. Automata that have predicates on their transitions were already proposed in [45]. This initial idea has recently been expanded and more fully investigated in symbolic automata [47, 46, 20]. 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 [19], 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.

We initially presented the results regarding *SRT* in [10] (we called them Register Match Automata in that report). The difference between that report and the present paper is that now we use a different formalism for expressing patterns at the language level. However, the automaton model remains essentially the same. Automaton models similar to *SRT* have been independently presented in [19] and [11]. In both cases, the focus was on Symbolic Register Automata, i.e., on automata without any output on their transitions. The former work focused on an extensive theoretical analysis, while the latter on the theoretical applicability of this type of automata for CER, without presenting an implementation.

2.2 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, this is not the case. SASE 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. Cayuga attempts to address these issues of constraints on its expressive power through the method of resubscription, i.e., expressions which cannot be captured by a single automaton are compiled into multiple automata [21]. 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. Our system does not suffer from these limitations. Its novelty is that it provides formal, compositional semantics which allows us to address all of the above issues. We show that negation is the only problematic operator. The 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 [28, 29] and Wayeb [12, 9] 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.

2.3 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 [16]. These automata are called augmented finite automata (AFA) and are enriched with registers, in order to capture trends. With respect to compositionality, AFA are similar to *SRT*: Like *SRT*, Augmented Finite Automata (AFA) [16] 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*. AFA are thus less expressive than *SRT*. Additionally, AFA are not transducers and cannot enumerate the input events of a complex event. They can report event lifetimes, i.e., the duration of a complex event. *SRT* can also report individual input events. The input events can be reconstructed in a port-processing step, if needed, from the lifetime, but this seems to hold only for contiguous patterns. It is unclear whether this is feasible for non-contiguous patterns. 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, 42]. 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 [50, 35]. 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 and to what extent the proposed optimizations would work for patterns processing events in real time.

3 Symbolic Regular Expressions with Memory and Output

The field of CER has been growing strong for the past 20 years. It is thus no surprise that there is no lack of languages, formalisms and systems from which one may choose according to their needs. As a result, there is considerable variability concerning the most relevant and useful operators of CER patterns, their semantics and the corresponding computational models to be used for the actual detecting of complex events. On the one hand, this variability may be viewed as a sign of vigor for the field. On the other hand, the fact that operators and their semantics are sometimes defined informally makes it hard to compare different systems in terms of their expressive capabilities. It also makes it hard to study a single system in itself in a more systematic manner, other than actually running it and observing its behavior.

As an attempt to mitigate these problems, we present and describe a framework for CER which has formal, denotational semantics. We first present 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 number of cells in a simulated tumor at the current timepoint is higher than their number at the previous timepoint. We also discuss the semantics

of patterns written in our proposed language and show that these are well-defined. As a result, in order to know whether a given stream contains any 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. Not only is this feature critical in itself, allowing for a systematic understanding of the use of operators, but it could also 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. Previous work on CER has produced systems which are highly expressive (e.g., FlinkCEP [4]), but lack a proper, formal description. Some more recent work ([15]) has attempted to construct a system which is both formal and efficient. However, it does not support n-ary expressions, allowing (non-temporal) constraints which are applied only to the last event read from a stream.

Before presenting *SRT*, we first present a high-level formalism for defining CER patterns. We extend the work presented in [36], 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 against a new input element for (in)equality. 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*).

First, in Section 3.1 we repeat some basic definitions from logic theory. We also describe how we can adapt them and simplify them to suit our needs. Next, in Section 3.2 we precisely define the notion of conditions. In *SREMO*, conditions will act in a manner equivalent to that of terminal symbols in classical regular expressions. The difference is of course that conditions are essentially logic formulas that can reference both the current element read from a string/stream and possibly some past elements. In Section 3.3 we present the syntax for *SREMO* and in Section 3.4 the definition of their semantics.

3.1 Formulas and models

In this section, we follow the notation and notions presented in [31]. The first notion that we need is that of a $\mathcal{V}$-structure. A $\mathcal{V}$-structure essentially describes a domain along with the operations that can be performed on the elements of this domain and their interpretation.

▶ **Definition 2** ($\mathcal{V}$-structure [31]). *A vocabulary $\mathcal{V}$ is a set of function, relation and constant symbols. A $\mathcal{V}$-structure is an underlying set $\mathcal{U}$, called a universe, and an interpretation of $\mathcal{V}$. An interpretation assigns an element of $\mathcal{U}$ to each constant in $\mathcal{V}$, a function from $\mathcal{U}^n$ to $\mathcal{U}$ to each n-ary function in $\mathcal{V}$ and a subset of $\mathcal{U}^n$ to each n-ary relation in $\mathcal{V}$.* ◀

▶ **Example 3.** Using Example 1, we can define the following vocabulary

$$\mathcal{V} = \{R, c_1, c_2, c_3, c_4, c_5, c_6\}$$

and the universe

$$\mathcal{U} = \{(B,1,22,300), (B,1,24,225), (B,2,32,1210), (S,1,70,760), (S,1,68,2000), (B,2,33,95)\}$$

We can also define an interpretation of V by assigning each c_i to an element of $\mathcal{U}$, e.g., c_1 to $(B,1,22,300)$, c_2 to $(B,1,24,225)$, etc. R may also be interpreted as $R(x,y) := x.id = y.id$, i.e., this binary relation contains all pairs of $\mathcal{U}$ which have the same id. For example, $((B,1,22,300),(S,1,70,760)) \in R$ and $((B,1,22,300),(B,2,33,95)) \notin R$. If there are more

(even infinite) tuples in a stream/string, then we would also need more constants (even infinite).

We 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 now define the syntax and semantics of formulas that can be constructed from the constants, relations and functions of a $\mathcal{V}$-structure. We begin with the definition of terms.

▶ **Definition 4** (Term [31]). *A term is defined inductively as follows:*
- *Every constant is a term.*
- *If f is an m-ary function and $t_1, \cdots, t_m$ are terms, then $f(t_1, \cdots, t_m)$ is also a term.* ◀

Using terms, relations and the usual Boolean constructs of conjunction, disjunction and negation, we can define formulas.

▶ **Definition 5** (Formula [31]). *Let t_i be terms. A formula is defined as follows:*
- *If P is an n-ary relation, then $P(t_1, \cdots, t_n)$ is a formula (an atomic formula).*
- *If ϕ is a formula, $\neg\phi$ is also a formula.*
- *If ϕ_1 and ϕ_2 are formulas, $\phi_1 \wedge \phi_2$ is also a formula.*
- *If ϕ_1 and ϕ_2 are formulas, $\phi_1 \vee \phi_2$ is also a formula.* ◀

▶ **Definition 6** ($\mathcal{V}$-formula [31]). *If $\mathcal{V}$ is a vocabulary, then a formula in which every function, relation and constant is in $\mathcal{V}$ is called a $\mathcal{V}$-formula.* ◀

▶ **Example 7.** Continuing with our example, $R(c_1, c_4)$ is an atomic $\mathcal{V}$-formula. $R(c_1, c_4) \wedge \neg R(c_1, c_3)$ is also a (complex) $\mathcal{V}$-formula, where $\mathcal{V} = \{R, c_1, c_2, c_3, c_4, c_5, c_6\}$.

Notice that in typical definitions of terms and formulas (as found in [31]) variables are also present. A variable is also a term. Variables are also used in existential and universal quantifiers to construct formulas. In our case, we will not be using variables in the above sense (instead, as explained below, we will use variables to refer to registers). Thus, existential and universal formulas will not be used. In principle, they could be used, but their use would be counter-intuitive. At every new event, we need to check whether this event satisfies some properties, possibly in relation to previous events. A universal or existential formula would need to check every event (variables would refer to events), both past and future, to see if all of them or at least one of them (from the universe $\mathcal{U}$) satisfy a given property. Since we will not be using variables, there is also no notion of free variables in formulas (variables occurring in formulas that are not quantified). Thus, every formula is also a sentence, since sentences are formulas without free variables. In what follows, we will thus not differentiate between formulas and sentences.

We can now define the semantics of a formula with respect to a $\mathcal{V}$-structure.

▶ **Definition 8** (Model of $\mathcal{V}$-formulas [31]). *Let $\mathcal{M}$ be a $\mathcal{V}$-structure and ϕ a $\mathcal{V}$-formula. We define $\mathcal{M} \models \phi$ ($\mathcal{M}$ models ϕ) as follows:*
- *If ϕ is atomic, i.e. $\phi = P(t_1, \cdots, t_m)$, then $\mathcal{M} \models P(t_1, \cdots, t_m)$ iff the tuple $(a_1, \cdots, a_m)$ is in the subset of $\mathcal{U}^m$ assigned to P, where a_i are the elements of $\mathcal{U}$ assigned to the terms t_i.*

- *If* $\phi := \neg\psi$*, then* $\mathcal{M} \models \phi$ *iff* $\mathcal{M} \not\models \psi$.
- *If* $\phi := \phi_1 \wedge \phi_2$*, then* $\mathcal{M} \models \phi$ *iff* $\mathcal{M} \models \phi_1$ *and* $\mathcal{M} \models \phi_2$.
- *If* $\phi := \phi_1 \vee \phi_2$*, then* $\mathcal{M} \models \phi$ *iff* $\mathcal{M} \models \phi_1$ *or* $\mathcal{M} \models \phi_2$. ◀

▶ **Example 9.** If $\mathcal{M}$ is the $\mathcal{V}$-structure of our example, then $\mathcal{M} \models R(c_1, c_4)$, since $c_1 \rightarrow (B, 1, 22, 300)$, $c_4 \rightarrow (S, 1, 70, 760)$ and $((B, 1, 22, 300), (S, 1, 70, 760)) \in R$. We can also see that $\mathcal{M} \models R(c_1, c_4) \wedge \neg R(c_1, c_3)$, since $c_3 \rightarrow (B, 2, 32, 1210)$ and $((B, 1, 22, 300), (B, 2, 32, 1210)) \notin R$.

3.2 Conditions

Based on the above definitions, we will now define conditions over registers. 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 10** (Condition). *Let* $\mathcal{M}$ *be a* $\mathcal{V}$*-structure always equipped with the unary relation* $\top$ *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 essentially a* $\mathcal{V}$*-formula, as defined above (Definition 5), where, instead of terms, we use 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 11.** 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. We will show later how registers are written.

Since terms now refer to registers, we need a way to access the contents of these registers. We will assume that each register has the capacity to store exactly one element from $\mathcal{U}$. The notion of valuations provides us with a way to access the contents of registers.

▶ **Definition 12** (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". The set of all valuations on* R *is denoted by* $F(r_1, \cdots, r_k)$*. Register update happens with* $v[r_i \leftarrow u]$*, denoting the valuation where we replace the content of* r_i *with a new element* u*, producing a new valuation* v'*:*

$$v'(r_j) = v[r_i \leftarrow u] = \begin{cases} u & \text{if } r_j = r_i \\ v(r_j) & \text{otherwise} \end{cases} \tag{1}$$

Similarly, $v[W \leftarrow u]$, where $W \subseteq R$, denotes the valuation obtained by replacing the contents of all registers in W with u. We say that a valuation v is compatible *with a condition ϕ if, for every register variable r_i that appears in ϕ, $v(r_i)$ is defined, i.e., r_i is not 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 will denote the "contents" of the automaton's head, i.e., the last element read from the string.* ◀

A valuation v is essentially a function with which we can retrieve the contents of any register.

We can now define the semantics of conditions, similarly to the way we defined models of $\mathcal{V}$-formulas in Definition 8. The difference is that the arguments to relations are no longer elements assigned to terms but elements stored in registers, as retrieved by a given valuation.

▶ **Definition 13** (Semantics of conditions). *Let $\mathcal{M}$ be a $\mathcal{V}$-structure, $u \in \mathcal{U}$ an element of the universe of $\mathcal{M}$ and $v \in F(r_1, \cdots, r_k)$ a valuation. We say that a condition ϕ is satisfied by (u, v), denoted by $(u, v) \models \phi$, iff one of the following holds:*

- *$\phi := \top$, i.e., $(u, v) \models \top$ for every element and valuation.*
- *$\phi := P(x_1, \cdots, x_n)$, $x_i \in R \cup \{\sim\}$, $v(x_i)$ is defined for all x_i and $u \in P(v(x_1), \cdots, v(x_n))$.*
- *$\phi := \neg\psi$ and $(u, v) \not\models \psi$.*
- *$\phi := \phi_1 \wedge \phi_2$, $(u, v) \models \phi_1$ and $(u, v) \models \phi_2$.*
- *$\phi := \phi_1 \vee \phi_2$, $(u, v) \models \phi_1$ or $(u, v) \models \phi_2$.* ◀

▶ **Example 14.** Returning to our example, we can check whether the condition $\phi_1 := TypeIsB(\sim)$ is satisfied by the first element of Table 1, $(B, 1, 22, 300)$. We assume that we start with a valuation v where all registers are empty. Indeed $((B, 1, 22, 300), v) \models \phi_1$, since $v(\sim)$ is defined and $(B, 1, 22, 300) \in TypeIsB(v(\sim))$. Note that $v(\sim)$ is always defined because the automaton head always points to an element. The only exception is when we are at the very beginning of a string, without having read any elements.

3.3 SREMO syntax

We are now in a position to define Symbolic Regular Expressions with Memory and Output *SREMO*. We achieve this 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 $\bullet$, indicating that the event must be marked as being part of the complex event, or $\otimes$, indicating that the event is irrelevant and should be excluded from any detected complex events.

▶ **Definition 15** (Symbolic regular expression with memory and output (*SREMO*)). *A symbolic regular expression with memory and output over a $\mathcal{V}$-structure $\mathcal{M}$ and a set of register variables $R = \{r_1, \cdots, r_k\}$ is inductively defined as follows:*

1. *ϵ and $\emptyset$ are SREMO.*
2. *If ϕ is a condition (as in Definition 10) 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. This is the case where we need to store the current element read from the automaton's head to register r_i.*
4. *If e_1 and e_2 are SREMO, then $e_1 + e_2$ is also a SREMO. This corresponds to disjunction.*
5. *If e_1 and e_2 are SREMO, then $e_1 \cdot e_2$ is also a SREMO. This corresponds to concatenation.*
6. *If e is a SREMO, then e^* is also a SREMO. This corresponds to Kleene-star.* ◀

ϵ is the regular expression (known from classical automata) 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. We will see later if and under which requirements other possible operators, like intersection and negation, may also be added to *SREMO*.

▶ **Example 16.** We now have everything we need to express the pattern of our example. Consider the following *SREMO*:

$$\begin{aligned} e_1 :=& (\mathit{TypeIsB}(\sim) \uparrow \bullet \downarrow r_1) \cdot (\top \uparrow \otimes)^* \cdot \\ & ((\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \uparrow \bullet) \end{aligned} \tag{2}$$

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 this event as part of the match as well.

3.4 SREMO semantics

In order to define the semantics of *SREMO*, we need to define how the contents of the registers may change. 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 $i_k \leq n$. 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 $max(M_1) < min(M_2)$.

▶ **Definition 17** (Semantics of *SREMO*). *Let e be a SREMO over a $\mathcal{V}$-structure $\mathcal{M}$ and a set of register variables $R = \{r_1, \cdots, r_k\}$, S a string constructed from elements of the universe of $\mathcal{M}$, M a candidate match of e on S and $v, v' \in F(r_1, \cdots, r_k)$. We define the relation $(e, S, M, v) \vdash v'$ as follows:*

1. *$(\epsilon, S, M, v) \vdash v'$ iff $S = \epsilon$ and $v = v'$ (by definition, $M = \emptyset$).*
2. *$(\phi \uparrow o, S, M, v) \vdash v'$ iff $\phi \neq \epsilon$, $S = u$, $(u, v) \models \phi$, $v' = v$ and*

$$\begin{cases} o = \otimes \text{ and } M = \emptyset & \text{or} \\ o = \bullet \text{ and } M = \{i_u\} \end{cases}$$

   where i_u is the index of u.
3. *$(\phi \uparrow o \downarrow r_i, S, M, v) \vdash v'$ iff $\phi \neq \epsilon$, $S = u$, $(u, v) \models \phi$, $v' = v[r_i \leftarrow u]$ and*

$$\begin{cases} o = \otimes \text{ and } M = \emptyset & \text{or} \\ o = \bullet \text{ and } M = \{i_u\} \end{cases}$$

.

4. $(e_1 \cdot e_2, S, M, v) \vdash v'$ *iff* $S{=}S_1 \cdot S_2$ *and* $M{=}M_1 \cdot M_2$*:* $(e_1, S_1, M_1, v) \vdash v''$ *and* $(e_2, S_2, M_2, v'') \vdash v'$.
5. $(e_1 + e_2, S, M, v) \vdash v'$ *iff* $(e_1, S, M, v) \vdash v'$ *or* $(e_2, S, M, v) \vdash v'$.
6. $(e^*, S, v) \vdash v'$ *iff*

$$\begin{cases} S = \epsilon \text{ and } v' = v & \text{or} \\ S{=}S_1 \cdot S_2, M{=}M_1 \cdot M_2 : (e, S_1, M_1, v) \vdash v'' \text{ and } (e^*, S_2, M_2, v'') \vdash v' \end{cases}$$

◀

In the first case, we have an ϵ *SREMO*. It may reach another valuation only if it reads an ϵ string and this new valuation is the same as the initial one, i.e., the registers do not change. In the second case, where we have a condition $\phi \neq \epsilon$, we move to a new valuation only if the condition is satisfied with the current element u and the given register contents v. Again, the registers do not change. Additionally, if $o = \bullet$, we accept the current element (its index i_u) as part of the match M. Otherwise, we ignore it. The third case is similar to the second, with the important difference that the register r_i needs to change and to store the current element. For the fourth case (concatenation), we need to be able to break the initial string into two sub-strings such that the first one reaches a certain valuation and the second one can start from this new valuation and reach another one. Similarly, the fifth case represents a disjunction of *SREMO*. Finally, the sixth case (iteration) requires that we break the initial string into multiple sub-strings such that each one of these sub-strings can reach a valuation and the next one can start from this valuation and reach another one.

Based on the above definition, 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 18** (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 19.** We can now continue with our example. If we feed the string/stream of Table 1 to *SREMO* (2) of Example 16, then we will have the following. First, we apply case (4) of Definition 17. We have

$$e_1 := (\mathit{TypeIsB}(\sim) \uparrow \bullet \downarrow r_1)$$

and

$$e_2 := (\top \uparrow \otimes)^* \cdot ((\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \uparrow \bullet)$$

We break the string S of Table 1 into two sub-strings S_1 and S_2, where S_1 is the first element of S, $(B, 1, 22, 300)$, and S_2 the remaining five. We check whether S_1 satisfies e_1, by applying case (3) of Definition 17. Since the type of S_1 is B (BUY), we will move on and store $(B, 1, 22, 300)$ to register r_1, i.e., we will move from the empty valuation where $v(r_1) = \sharp$ to v', where $v'(r_1) = (B, 1, 22, 300)$. We will also "accept" this element as part of a potential future match. We then check e_2 and S_2. We apply again case (4) of Definition 17 and break e_2 into $(\top \uparrow \otimes)^*$ and $((\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \uparrow \bullet)$. Then, the sub-expression

$(\top \uparrow \otimes)^*$ lets us skip and ignore any number of elements. We can thus skip the second and third elements without changing the register contents. Now, upon reading the fourth element $(S, 1, 70, 760)$, there are two options. Either skip it again to read the fifth element or try to move on by checking the sub-expression $(\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1) \uparrow \bullet)$. This latter condition is actually satisfied, since the type of this element is indeed S and its *id* is equal to the *id* of the element store in r_1. Thus, $S_{1..4}$ is indeed accepted by e_1. $M = \{1, 4\}$ is also a match of e_1 on S (and on $S_{1..4}$). With a similar reasoning we can see that the same is also true for $S_{1..5}$ and $M = \{1, 5\}$, had we chosen to skip the fourth element.

It can be shown that the concept of matches "subsumes" that of languages, i.e., if two *SREMO* have the same matches for every string S, then they also have the same languages.

▶ **Theorem 20.** *Let e, e' be two SREMO. If, for every string S, $\mathit{Match}(e, S) = \mathit{Match}(e', S)$, then $\mathit{Lang}(e) = \mathit{Lang}(e')$.*

Proof. The proof may be found in the Appendix, see A.2. ◀

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

We now show how *SREMO* can be translated to an appropriate automaton model and how this model may then be used to perform CER. We also study the closure properties of this automaton model.

4.1 Definition of 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 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.

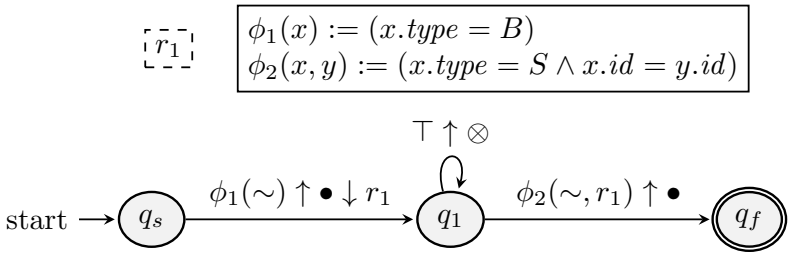


Figure 1 *SRT* corresponding to the *SREMO* of eq. (2).

We now formally define *SRT*. To aid understanding, we present three separate definitions: one for the automaton itself, one for its configurations and one for its runs. 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 21** (Symbolic Register Transducer). *A symbolic register transducer (SRT) with k registers over a $\mathcal{V}$-structure $\mathcal{M}$ 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 10 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 22.** 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 a unary condition that 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. (2).

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

▶ **Definition 23** (Configuration of *SRT*). *Assume a string* $S = t_1, t_2, \cdots, t_l$ *and a SRT* T *consuming* S*. A configuration of* T *is a triple* $c = [j, q, v] \in \mathbb{N} \times Q \times F(r_1, \cdots, r_k)$*, where*

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

We say that $c' = [j', q', v']$ *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:

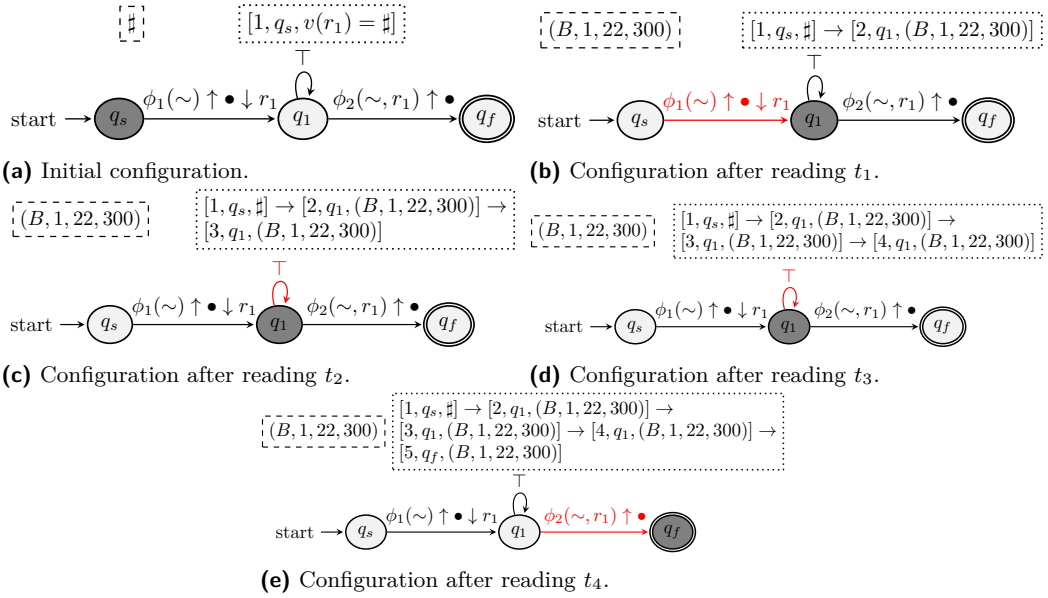
▶ **Definition 24** (Run of *SRT* over string/stream). *A run* ϱ *of a SRT* T *over a stream* $S = t_1, \cdots, t_n$ *is a sequence of successor configurations* $[1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} [n + 1, q_{n+1}, v_{n+1}]$*. A run 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 25.** An accepting run of the *SRT* of Figure 1, while consuming the first four events from the stream of Table 1, is the following:

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

Transition subscripts in this example refer to states of the *SRT*, e.g., $\delta_{s,s}$ is the transition from the start state to itself, $\delta_{s,1}$ is the transition from the start state to q_1, etc. Note that



(a) Initial configuration.

(b) Configuration after reading t_1.

(c) Configuration after reading t_2.

(d) Configuration after reading t_3.

(e) Configuration after reading t_4.

Figure 2 A run of the *SRT* of Figure 1, while consuming the first four events from the stream of Table 1. Triggered transitions are shown in red and the current state of the *SRT* in dark gray. The dashed box represents a register. The contents of the register at each configuration are shown inside the dashed box. Inside the dotted boxes, the run is shown.

the valuation (contents or register r_1) changes only once, from ♯ (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. See also Figure 2. Run (3) 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 26** (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 $v_1 = ♯$. 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 $Match(T, S) = \{Match(\varrho) \mid \varrho \in Run_f(T, S)\}$.* ◀

4.2 Properties of SRT

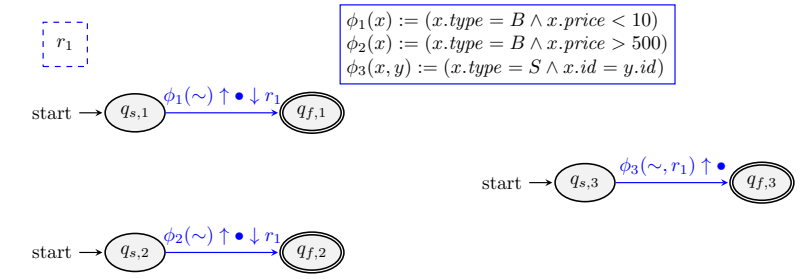
We now study the properties of *SRT*. First, we prove that *SREMO* can be compiled to *SRT*. We then show that *SRT* are closed under union, intersection, concatenation and Kleene-start but not under complement and determinization. We can thus construct *SREMO* and *SRT* by using arbitrarily (in whatever order and depth is required) the four basic operators of union, intersection, concatenation and Kleene-star. However, the negative result about complement suggests that the use of *negation* in CER patterns cannot be equally arbitrary. Moreover,

deterministic *SRT* cannot be used in cases where this might be required, as in Complex Event Forecasting [12]. If, however, we use an extra window operator, effectively limiting the length of strings accepted by a *SRT*, we can then show that closure under complement and determinization is also possible.

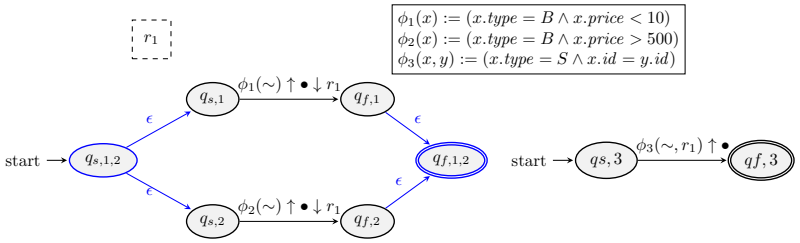
We first prove that, for every *SREMO* there exists an equivalent *SRT*. The proof is constructive, similar to that for classical automata. Equivalence between an expression e and a *SRT* T means that they recognize the same language and have the same matches. See Definitions 18 and 26.

▶ **Theorem 27.** *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.*

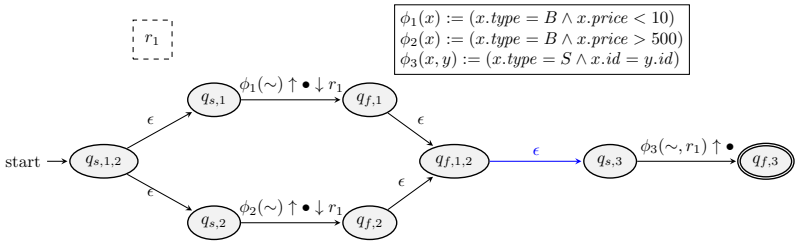
Proof. The complete construction process and proof may be found in Appendix A.2. ◀



(a) Constructing *SRT* for terminal sub-expressions.



(b) Connecting *SRT* via disjunction.



(c) Connecting *SRT* via concatenation.

Figure 3 Constructing *SRT* from *SREMO* (4). New elements added at every step are shown in blue.

▶ **Example 28.** Here, we present an example, to give the intuition. Let

$$e_2 := ((\phi_1(\sim) \uparrow \bullet \downarrow r_1) + (\phi_2(\sim) \uparrow \bullet \downarrow r_1)) \cdot (\phi_3(\sim, r_1) \uparrow \bullet) \tag{4}$$

be a *SREMO*, where

$$\phi_1(x) := (x.type = B \wedge x.price < 10)$$
$$\phi_2(x) := (x.type = B \wedge x.price > 500)$$
$$\phi_3(x, y) := (x.type = S \wedge x.id = y.id)$$

With this expression, we want to monitor stocks for possible suspicious transactions. We want to detect cases where a stock is initially bought at a very low or high price (first line of *SREMO* (4)) and then sold. The last condition is a binary formula, applied to both $\sim$ and r_1. It ensures that matches refer to the same company. Figure 3 shows the process for constructing the *SRT* which is equivalent to *SREMO* (4).

The algorithm is compositional, starting from the base cases $e{:=}\phi \uparrow o$ or $e{:=}\phi \uparrow o \downarrow W$. The three regular expression operators (concatenation, disjunction, Kleene-star) are handled in a manner almost identical as for classical automata. The subtlety here concerns the handling of registers. The simplest solution is to gather from the very start all registers mentioned in any sub-expressions of the original *SREMO* e, i.e. any registers in the register selection of any transitions and any write registers. We first create those registers and then start the construction of the sub-automata. Note that some registers may be mentioned in multiple sub-expressions (e.g., in one that writes to it and then in one that reads its contents). We only add such registers once. We treat the registers as a set with no repetitions.

For the example of Figure 3, only one register is mentioned, r_1. We start by creating this register. Then, we move on to the terminal sub-expressions. There are three basic sub-expressions and three basic automata are constructed: from $q_{s,1}$ to $q_{f,1}$, from $q_{s,2}$ to $q_{f,2}$ and from $q_{s,3}$ to $q_{f,3}$. See Figure 3a. To the first two transitions, we add the relevant *unary* conditions, e.g., we add $\phi_1(x){:=}(x.type{=}B \wedge x.price{<}10)$ to $q_{s,1}{\rightarrow}q_{f,1}$. To the third transition, we add the relevant *binary* condition $\phi_3(x, y) := (x.type = S \wedge x.id = y.id)$. The + operator is handled by joining the *SRT* of the disjuncts through new states and ϵ-transitions. See Figure 3b. The concatenation operator is handled by connecting the *SRT* of its sub-expressions through an ϵ-transition, without adding any new states. See Figure 3c. Iteration, not applicable in this example, is handled by joining the final state of the original automaton to its start state through an ϵ-transition.

The standard result about ϵ elimination also holds, stating that we can always eliminate all ϵ transitions from a *SRT* to get an equivalent *SRT* with no ϵ transitions.

▶ **Lemma 29.** *For every SRT T_ϵ with ϵ transitions there exists an equivalent SRT $T_{\notin}$ without ϵ transitions, i.e., a SRT such that $Match(T_\epsilon, S) = Match(T_{\notin}, S)$ for every string S.*

Proof. See Appendix A.3. ◀

We now study the closure properties of *SRT* under union, concatenation and Kleene-star. We give the definition for closure under these operations:

▶ **Definition 30** (Closure of *SRT*). *We say that SRT are closed under:*

- *union if, for every SRT T_1 and T_2, there exists a SRT T such that $Match(T, S) = Match(T_1, S) \cup Match(T_2, S)$, i.e., M is a match of T iff it is a match of T_1 or T_2.*
- *concatenation if, for every SRT T_1 and T_2 and strings S_1, S_2, there exists a SRT T such that $Match(T, S) = Match(T_1, S_1) \cdot Match(T_2, S_2)$, where $S = S_1 \cdot S_2$, i.e., M is a match of T iff M_1 is a match of T_1, M_2 is a match of T_2 and M is the concatenation of M_1 and M_2 (i.e., $M = M_1 \cup M_2$ and $min(M_2) > max(M_1)$).*

- *Kleene-star if, for every SRT T and string S, there exists a SRT T_* such that $Match(T_*, S) = \{M : M = M_1 \cdot M_2 \cdots M_n, M_i = Match(T, S_i), S = S_1 \cdot S_2 \cdots S_n\}$, i.e., M is a match of T_* iff each M_i is a match of T and M is the concatenation of all M.*

◀

We thus have the following for union, concatenation and Kleene-star:

▶ **Theorem 31.** *SRT are closed under union, concatenation and Kleene-star.*

Proof. See Appendix A.4. ◀

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

4.3 Streaming symbolic register transducers

We have thus far described how *SREMO* and *SRT* can be applied to bounded strings that are known in their totality before recognition. A string is given to a *SRT* and an answer is expected about whether the whole string belongs to the automaton's language or not along with any matches detected. However, in CER we are required to handle continuously updated streams of events and detect instances of *SREMO* satisfaction as soon as they appear in a stream. For example, the automaton of the classical regular expression $a \cdot b$ would accept only the string a, b. In a streaming setting, we would like the automaton to report a match every time this string appears in a stream. For the stream a, b, c, a, b, c, we would thus expect two matches to be reported, one after the second symbol and one after the fifth (assuming that we are interested only in contiguous matches).

Slight modifications are required so that *SREMO* and *SRT* may work in a streaming setting (the discussion in this section develops along the lines presented in our previous work [12], with the difference that here we are concerned with symbolic automata with memory and output). First, we need to make sure that the automaton can start its recognition after every new element. If we have a classical regular expression R, we can achieve this by applying on the stream the expression $\Sigma^* \cdot R$, where Σ is the automaton's (classical) alphabet. For example, if we apply $R := \{a, b, c\}^* \cdot (a \cdot b)$ on the stream a, b, c, a, b, c, the corresponding automaton would indeed reach its final state after reading the second and the fifth symbols. In our case, events come in the form of tuples with both numerical and categorical values. Using database systems terminology we can speak of tuples from relations of a database schema [28]. These tuples constitute the universe $\mathcal{U}$ of a $\mathcal{V}$-structure $\mathcal{M}$. A stream S then has the form of an infinite sequence $S = t_1, t_2, \cdots$, where $t_i \in \mathcal{U}$. Our goal is

- first, to report the indices i at which a complex event is detected;
- second, to report the indices of the simple events from which a complex event is composed;
- while taking into account the fact that, at a given index i, multiple complex events may be detected.

More precisely, if $S_{1..k} = \cdots, t_{k-1}, t_k$ is the prefix of S up to the index k, we say that a *SREMO* e is detected at k iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in Lang(e)$. Additionally, the streaming matches detected at k are defined as $Match_{stream}(e, S) = \{M : M \in Match(e, S_{m..k}) \; \forall \; m, 1 \leq m \leq k\}$

In order to detect complex events of a *SREMO* e on a stream, we use a streaming version of *SREMO* and *SRT*.

▶ **Definition 32** (Streaming *SREMO* and *SRT*). *If e is a SREMO, then $e_s = \top^* \cdot e$ is called the streaming SREMO (sSREMO) corresponding to e. A SRT T_{e_s} constructed from e_s is called a streaming SRT (sSRT) corresponding to e.* ◀

Using $e_s = \top^* \cdot e$ we can detect complex events of e while reading a stream S, since a stream segment $S_{m..k}$ belongs to the language of e iff the prefix $S_{1..k}$ belongs to the language of e_s. The prefix $\top^*$ lets us skip any number of events from the stream and start recognition at any index $m, 1 \leq m \leq k$.

5 Closure properties and selection strategies

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/conjunction and Kleene-star/iteration, CER systems make extensive use of other operators as well and even constructs which are external to the language itself. In this Section, we focus on the issue of how and if our proposed framework can accommodate these extra operators and constructs. We specifically discuss the following aspects of CER which are very common in the literature, but have been excluded from our presentation thus far: the operators of intersection/conjunction and complement/negation, the possibility of using deterministic automata for CER, the use of windows and the semantics of selection strategies,

5.1 Intersection and complement

We first study the closure properties of *SRT* under intersection and complement, two popular operators in CER.

The formal definition of closure under intersection and complement is as follows:

▶ **Definition 33** (Closure of *SRT*(intersection, complement)). *We say that SRT are closed under:*

- *intersection if, for every SRT T_1 and T_2, there exists a SRT T such that $Match(T, S) = Match(T_1, S) \cap Match(T_2, S)$, i.e., M is a match of T iff it is a match of T_1 and T_2.*
- *complement if, for every SRT T, there exists a SRT T_c such that for every string S it holds that $M \in Match(T, S) \Leftrightarrow M \notin Match(T_c, S)$.*

◀

With regards to intersection, we can prove the following:

▶ **Theorem 34.** *SRT are closed under intersection.*

Proof. See Appendix A.4. ◀

Note that intersection was not defined as an operator of *SREMO* in Definition 15. Theorem 34 indicates that we can introduce such an operator without any difficulties. It is important to distinguish intersection from another operator in CER, which is also often called conjunction and whose intended semantics is that a sequence of events must occur, regardless of their temporal order. This conjunction operator does not require any special treatment, as it can be readily expressed in *SREMO* by combining the already available operators of sequence and disjunction. For example, if we use $*$ to denote that type of conjunction, then we could write $*(\phi_1, \phi_2) := (\phi_1 \cdot \phi_2) + (\phi_2 \cdot \phi_1)$.

On the other hand, as is the case for register automata [34], *SRT* are not closed under complement:

▶ **Theorem 35.** *SRT are not closed under complement.*

Proof. See Appendix A.5. ◀

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. There is a subtle difference between negation as a regular expression operator and negation as a logical operator allowed in conditions (in Definition 10). Logical negation always requires that an event occurs and that it does not satisfy the negated condition. Regular negation may be "satisfied" even in the absence of any events and this is the way it is mostly used in CER patterns.

However, we can (partially) overcome the negative results about negation by using windows in *SREMO* and *SRT*, i.e., by limiting the length of strings accepted by *SREMO* and *SRT*. The general idea is that windows allow us to determinize *SRT*. With a deterministic *SRT* at hand, we can easily construct its complement. The downside is that we lose the ability to mark events that correspond to negated expressions. Thus, we now study the determinizability of *SRT*.

5.1.1 Determinization of SRT

In CER, it is typically the case that non-deterministic automata are employed because they can fully enumerate all the detected matches, i.e., report all input events comprising a match. We also use non-deterministic *SRT* as a computational model for CER because they can enumerate all the detected matches, i.e., report all input events comprising a match. Recall that complex events (or full matches) are defined as sets (of indices) of simple events. Non-deterministic *SRT* have the ability to create multiple runs as they consume a stream of events. Each run can mark different input events. Each run that reaches a final state can then report all the input events that it has marked. Thus, all complex events can be fully reported. However, deterministic automata are critical in certain applications, as in Complex Event Forecasting [12], where the goal is to forecast whether a complex event is expected to occur, without necessarily being interested in a complete enumeration. For this reason, we also study whether *SRT* are determinizable.

We can show that *SRT* are not closed under determinization, a result which might seem discouraging. We first provide the definition for deterministic *SRT*. Informally, a *SRT* is said to be deterministic if, at any time, with the same input element, it can follow no more than one transition. The formal definition is as follows:

▶ **Definition 36** (Deterministic *SRT* (*dSRT*))**.** *A SRT T with k registers $\{r_1, \cdots, r_k\}$ over a $\mathcal{V}$-structure $\mathcal{M}$ is deterministic if, for all transitions $q, \phi_1 \uparrow o_1 \downarrow W_1 \rightarrow q_1 \in T.\Delta$ and $q, \phi_2 \uparrow o_2 \downarrow W_2 \rightarrow q_2 \in T.\Delta$, if $q_1 \neq q_2$ then, for all $u \in \mathcal{M}.\mathcal{U}$ and $v \in F(r_1, \cdots, r_k)$, $(u, v) \models \phi_1$ and $(u, v) \models \phi_2$ cannot both hold, i.e.,*

- *Either $(u, v) \models \phi_1$ and $(u, v) \not\models \phi_2$*
- *or $(u, v) \not\models \phi_1$ and $(u, v) \models \phi_2$*
- *or $(u, v) \not\models \phi_1$ and $(u, v) \not\models \phi_2$.*

◀

In other words, from all the outgoing transitions from a given state q at most one of them can be triggered on any element u and valuation/register contents v. By definition, for a deterministic *SRT*, at most one run may exist for every string/stream.

We say that a *SRT* T is determinizable if there exists a *dSRT* T_D such that $Match(T, S) = Match(T_D, S)$ for every string S. This is a strong notion of equivalence. By definition, a deterministic *SRT* can have at most one run and thus at most one match for any string. Thus, equivalence based on the matches would be hard to achieve, since non-deterministic *SRT* typically have multiple runs, each tracking a (candidate) match. Another notion of equivalence which is more relaxed can be obtained by requiring that the languages of T and T_D are the same, effectively ignoring the output of the transitions. In terms of CER, ignoring transition outputs would still allow us to detect complex events, in the sense that we could report at every timepoint whether at least one match has been fully completed. On the other hand, we would not be able to say neither whether more than one such matches occurred nor report the simple events comprising a complex one.

▶ **Definition 37** (Output-agnostic *SRT*)**.** *A SRT T is output-agnostic determinizable if there exists a deterministic SRT T_D such that $Lang(T) = Lang(T_D)$.*

Even with this more relaxed requirement, it is not always possible to determinize *SRT*:

▶ **Theorem 38.** *Not every SRT is output-agnostic determinizable.*

Proof. The proof is by a counter example. Let T denote the *SRT* of Figure 4. It detects events of type B followed by events of type S with the same identifier. Between the two events, B and S, any other events may also occur, due to the presence of a self-loop on state q_1 with the TRUE condition. This self-loop is what makes this *SRT* non-deterministic. Whenever the *SRT* is in state q_1 and an event of type S arrives with the same identifier as the stored B event, the automaton has two options. Either move to the final state q_f or remain in state q_1. The self-loop on the start state q_s also makes the *SRT* non-deterministic. T thus accepts strings S that contain a B followed by a S, whose identifiers are equal, regardless of the length of S. Any number of irrelevant events may precede the fist B event.

Assume there exist a deterministic *SRT* T_d with k registers which is equivalent to T. Let

$$S = (B, 1)(S, 2)$$

be a string given to T_d. After reading $S_1 = (B, 1)$, A_d must store it in a register r_1 in order to be able to compare it when $(S, 2)$ arrives. Let

$$S' = (B, 1)(B, 3)(S, 2)$$

After reading $S'_1 = (B, 1)$, T_d must store it in the register r_1, since T_d is deterministic and follows a single run. Thus, it must have the exact same behavior after reading S_1 and S'_1. But we must also store $S'_2 = (S, 3)$ after reading it. Additionally, S'_2 must be stored in a different register r_2. We cannot overwrite r_1. If we did this and S'_1 were $(B, 2)$, then we would not be able to match $(B, 2)$ to $S'_3 = (S, 2)$ and $S' = (B, 2)(B, 3)(S, 2)$ would not be accepted. Now, let

$$S'' = \underbrace{(B, \cdots)(B, \cdots)\cdots(B, \cdots)}_{k+1 \text{ elements}}(S, 2)$$

With a similar reasoning, all of the first $k + 1$ elements of S'' must be stored after reading them. But this is a contradiction, as T_d can store at most k different elements. Therefore, there does not exist a deterministic *SRT* which is equivalent to T.

◀

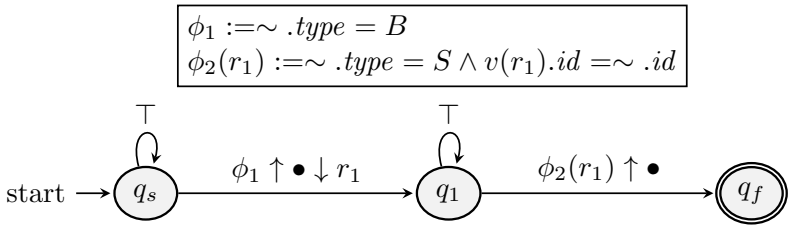


Figure 4 Example of a non-deterministic *SRT*.

This result could probably be generalized to state that *SREMO* cannot be captured by any deterministic automata with finite memory, at least if we retain the usual notions of determinism and memory. Determinism, in the sense that there can be at most one run of the automaton for every stream / string. Memory, in the sense that it can store a finite number of the input elements from the stream / string. We make this remark, because one can imagine finite memory structures that do not store elements. For example, a memory slot could store finite mathematical structures that could act as generators of a (possibly infinite) stream of past elements. Automata themselves are a typical case of a finite structure that can generate infinite sequences. Automata that act as recognizers and can store other automata, acting as generators, is thus something not inconceivable. Investigating such automata is, however, beyond the scope of this thesis.

5.1.2 Windowed SREMO/SRT

We can overcome the negative results about determinization by using windows in *SREMO* and *SRT*. We show that there exists a sub-class of *SREMO* for which a translation to output-agnostic deterministic *SRT* is indeed possible. This is achieved if we apply a windowing operator and limit 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. This observation motivates the introduction of windowing in *SREMO*.

▶ **Definition 39** (Windowed *SREMO*). *Let e be a SREMO over a $\mathcal{V}$-structure $\mathcal{M}$ and a set of register variables $R = \{r_1, \cdots, r_k\}$, S a string constructed from elements of the universe of $\mathcal{M}$ and $v, v' \in F(r_1, \cdots, r_k)$. A windowed SREMO (wSREMO) is an expression of the form $e_w := e^{[1..w]}$, where $w \in \mathbb{N}_1$. We define the relation $(e_w, S, M, v) \vdash v'$ as equivalent to: $(e, S, M, v) \vdash v'$ and $(max(M) - min(M) + 1) \leq w$.* ◀

Essentially, the only difference to regular *SREMO* is a slight change in the definition of the semantics (see Definition 17). For windowed *SREMO*, the additional requirement is that the "interval" from the smallest match index ($min(M)$) to the largest ($max(M)$) does not exceed the given window threshold.

▶ **Example 40.** If we apply a window $w = 4$ on *SREMO* (2), then $M = \{1, 4\}$ will still be a match, since $4 - 1 + 1 \leq 4$ obviously holds. $M = \{1, 5\}$, on the other hand, is no longer a match.

The windowing operator does not add any expressive power to *SREMO*. We could use the index of an event in the stream as an event attribute and then add binary conditions in an expression which ensure that the difference between the index of the last event read and

the first is no greater that w. It is more convenient, however, to have an explicit operator for windowing.

In order to derive a deterministic *SRT*, we can first construct a so-called "unrolled *SRT*" from a windowed expression, i.e., a *SRT* without any loops where each state may be visited at most once. The window allows us to do this, effectively removing any unbounded iterations. We can then apply a standard determinization algorithm to the unrolled *SRT*.

▶ **Theorem 41.** *For every windowed SREMO there exists an equivalent output-agnostic deterministic SRT.*

Proof. See Appendix A.7. ◀

▶ **Example 42.** As an example, consider Figure 5. Figure 5a shows the unrolled version of the *SRT* of Figure 4 for two different window values, 2 and 3. All cycles have been eliminated, with the overhead of one extra register being added. We also do not show the output, since we now focus on output-agnostic *SRT*. Note that the black *SRT* (for $w = 2$) is already deterministic. Figure 5b shows (part of) the deterministic *SRT* for $w = 3$. Due to space limitations, we show only part of the complete automaton. The idea is clear though. We start with the initial state (q_s) and create its mutually exclusive transitions. For example, if $(\phi_1 \wedge \top)$ (i.e., ϕ_1) evaluates to TRUE, then we move from q_s to both $q_{s,s}$ and $q_{s,1}$. We thus create a relevant hyper-state $\{q_{s,s}, q_{s,1}\}$ and connect it to the start state. We repeat this process until we have exhausted all possible states.

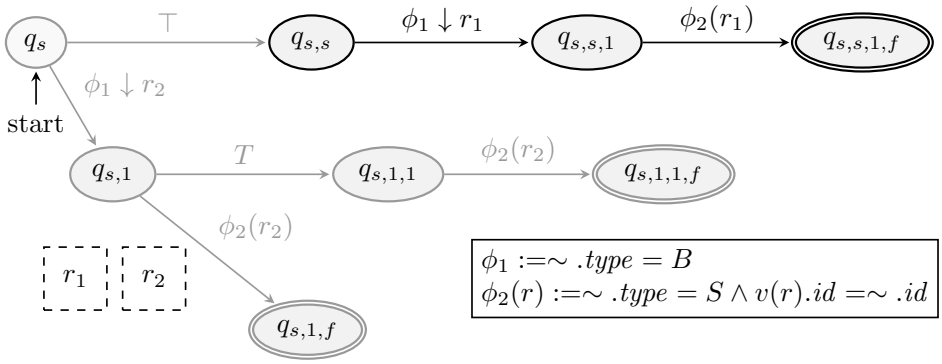
Deterministic *SRT* cannot thus be used for patterns without windows. By using windows, we can recover the property of *SRT* to be (output-agnostic) determinizable. The limitation at this point is that determinization is possible only at the language level (in Complex Event Forecasting, this does not constitute an issue).

If we restrict ourselves to windowed *SREMO* and to output-agnostic deterministic *SRT*, then we can prove that such *SRT* are closed under complement. A standard technique for creating the complement of an automaton is to create its deterministic equivalent and then flip its final states to non-final and vice versa. Thus, we may now prove, as a corollary, that windowed *SRT* are also closed under complement, when transition outputs are ignored:

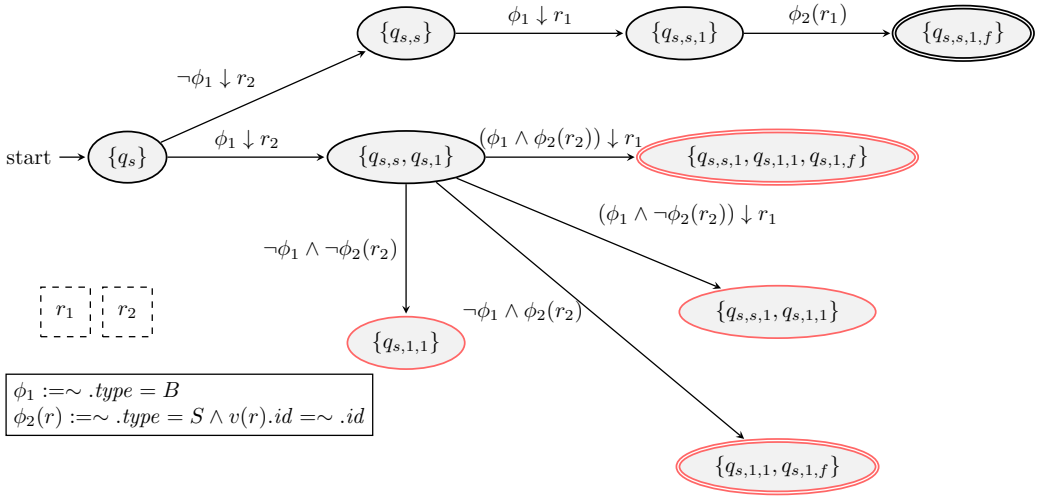
▶ **Corollary 43.** *Output-agnostic SRT compiled from windowed SREMO are closed under complement.*

Proof. See Appendix A.8. ◀

This result is important because it allows us to extend (windowed) *SREMO* so as to also include a negation operator. Although in theory the result about closure under complement holds only when outputs are ignored, in practice it could be useful even when we are indeed interested in the output of transitions and in marking some elements as belonging to a match. This could be the case when we have a *SREMO* containing a negation operator. Typically, we are not interested to mark any elements that are negated. Negation often implies that outputs should be ignored, especially when we want to imply absence of simple events, in which case there is no sense in marking those absent events. For example, consider the expression $((\phi_1(\sim) \uparrow \bullet \downarrow r_1) \cdot !(\phi_2) \cdot (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]}$, where ! stands for negation. In this case, we are only interested to mark the elements matching the first and third sub-expressions but not the second, negated one. We could construct a sub-automaton for the complement of ϕ_2, ignoring any outputs. At the same time, we could construct the automatons for the first and third sub-expressions as usual, with their outputs. By concatenating these three sub-automata, we would be able to properly mark the elements that we are interested in, despite the fact that the expression contains negation.



(a) *SRT* after unrolling cycles, for $w = 3$ (whole *SRT*, black and light gray states) and $w = 2$ (top 3 states in black).



(b) Output–agnostic deterministic *SRT*, for $w = 3$. Only part of it is shown. Red states can be further expanded.

Figure 5 Constructing a deterministic *SRT* from the *SRT* of Figure 4.

5.2 Selection strategies

CER patterns are usually characterized by their so-called selection strategy [27]. This strategy determines whether the input events in a match should occur contiguously in a stream (the standard interpretation of regular expressions) 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. strict-contiguity requires all simple events to occur contiguously. skip-till-any-match allows any irrelevant events to occur between the relevant ones. This behavior may typically be modeled in automata by introducing self-loops on states with $\top$ as their condition and $\otimes$ as their output (e.g., see Figure 1). On the other hand, skip-till-next-match also allows multiple irrelevant events between two relevant events, say S and B, except for B itself.

Given the properties of *SREMO* and *SRT*, the question is whether and which selection strategies may be accommodated, besides strict-contiguity, which is the standard interpretation of regular expressions. We can show that selection strategies may be applied as operators, through certain rewriting rules. This then implies that multiple and even nested strategies may be used in a pattern.

We define skip-till-any-match and skip-till-next-match as extra operators (and not extra-pattern constructs) in the following manner:

▶ **Definition 44** (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*

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

▶ **Definition 45** (skip-till-next-match). *If $e_1, e_2, \cdots, e_n$ are windowed SREMO, then $e_{next} := @(e_1, e_2, \cdots, e_n)$ is a SREMO, with @ denoting the skip-till-next-match selection strategy and*

$$e_{next} := e_1 \cdot (!e_2)^* \cdot e_2 \cdots (!e_n)^* \cdot e_n$$

! denotes the regular operator of negation/complement.

Similar definitions may be provided for selection strategies applied on iteration (Kleene-star or Kleene-plus), since iteration is also essentially sequential. With respect to disjunction, we make the assumption that a selection strategy applied on a disjunction operator has no effect. The strategy is not applied to any of the sub-expressions inside the top-level disjunction expression. If the user needs to apply a selection strategy to any sub-expression, he may do so by applying the relevant strategy operator on this specific disjunct / sub-expression.

The intuition behind the definition of skip-till-any-match is that we would like to be able to skip any events occurring between instances of e_i and e_{i+1}. We can actually achieve exactly this behavior by injecting between every pair of e_i and e_{i+1} the expression $(\top \uparrow \otimes)^*$. Since $\top$ evaluates to TRUE for every element, this means that $(\top \uparrow \otimes)^*$ allows us to skip any number of elements. These elements are skipped because that output of the expression is $\otimes$.

For skip-till-next-match, our goal is for an expression to exhibit a "greedy" behavior, i.e., after an instance of e_i we want to accept the immediately next instance of e_{i+1} and afterwards ignore any other instances of e_{i+1}. The above definition for skip-till-next-match satisfies this constraint. For example, consider the sub-expression $e_1 \cdot (!e_2)^* \cdot e_2$. Between e_1 and e_2 we have injected the sub-expression $(!e_2)^*$. This sub-expression ensures that, between instances of e_1 and e_2, no other instance of e_2 may occur. Thus, if after an instance of e_1, we encounter multiple instances of e_2, only the first one will be accepted.

It should be noted though that skip-till-next-match, in its most general form presented above, may be used only with windowed expressions. The reason is that it relies on negation, which, in turn, relies on determinization. A possible issue at this point is that windowed *SREMO* may be converted only to output-agnostic deterministic *SRT*. Thus, the deterministic sub-automata corresponding to the negated sub-expressions in skip-till-next-match (e.g., $(!e_2)^*$) do not have the ability to mark elements of the input string as relevant or irrelevant. However, this is not a serious limitation in this case. Since the negated sub-expressions are injected with the aim of skipping irrelevant events, we can simply force all transitions of these sub-automata to output $\otimes$, after we have constructed the automata. If, however, the sub-expressions e_i are terminal conditions ϕ_i (which is the typical case), then regular negation can be replaced with logical negation ($\neg$, as per Definition 10) and skip-till-next-match may then be used even in *SREMO* without windows.

As far as the implementation of selection strategies is concerned, we have included at the moment skip-till-any-match in our our system, but not yet skip-till-next-match. We have not had a need for the latter yet, which also depends on negation (also not implemented currently). We focused on skip-till-any-match which is the most demanding, both in terms of time and memory. Note that no special treatment is reserved for the selection strategies from an implementation point of view. A *SREMO* with a selection strategy applied to it treats

this strategy as another operator. The *SREMO* is first re-written according to Definitions 44 and 45. The re-written *SREMO* is then compiled into a *SRT*, just like every other *SREMO*. This *SRT* is then used for recognition, without any strategy-specific optimizations. Even in the absence of optimizations, we show that our system can handle even skip-till-any-match, the most relaxed and demanding strategy, due to its lightweight representation of runs.

5.3 Summary

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, but only for windowed expressions. When applied to simple conditions, it is available even for expressions without windows.

6 Implementation and Complexity

In the theory of formal languages it is customary to present complexity results for various decision problems, most commonly for the problem of non-emptiness (whether an expression or automaton accepts at least one string), that of membership (deciding whether a given string belongs to the language of an expression/automaton) and that of universality (deciding whether a given expression/automaton accepts every possible string). We briefly discuss here these problems for the case of *SREMO* and *SRT*.

The complexity of these problems for *SREMO* and *SRT* depends heavily on the nature of the conditions used as terminal expressions in *SREMO* and as transition guards in *SRT*. This, in turn, depends on the complexity of deciding whether a given element from the universe $\mathcal{U}$ of a $\mathcal{V}$-structure $\mathcal{M}$ belongs to a relation R from $\mathcal{M}$. Since we have not imposed until now any restrictions on such relations, the complexity of the aforementioned decision problems can be "arbitrarily" high and thus we cannot provide specific bounds. If, for example, the problem of evaluating a relation R is NP-complete and this relation is used in a *SREMO*/*SRT* condition, this then implies that the problem of membership immediately becomes at least NP-complete. In fact, if the problem of deciding whether an element from $\mathcal{U}$ belongs to a relation R is undecidable, then the membership problem becomes also undecidable.

We can, however, provide some rough bounds by looking at the complexity of these problems for the case of register automata (see [36]). Register automata are a special case of *SRT*, where the only allowed relations are the binary relations of equality and inequality and the transitions do not generate any output. We assume that these relations may be evaluated in constant time. For the problem of universality, we know that it is undecidable for register automata. We can thus infer that it remains so for *SRT* as well. On the other hand, the problem of non-emptiness is decidable but PSPACE-complete. The same problem for *SRT* is thus PSPACE-complete. Finally, the problem of membership is NP-complete. Therefore, it is also at least NP-complete for *SRT*. Note that membership is the most important problem for the purposes of CER, since in CER we continuously try to check whether a string (a suffix of the input stream) belongs to the language of a pattern's automaton. In general, if we assume that the problem of membership in all relations R is decidable in constant time, then the complexity of the decision problems for *SRT* coincides with that for register automata.

If we focus our attention even further on windowed *SRT*, as is the case in CER, then we can estimate more precisely the complexity of processing a single event from a stream. This is the most important operation for CER. A windowed *SRT* can first be determinized (offline) to obtain a *dSRT*. Assume that the resulting *dSRT* T has r registers and c conditions/minterms. We also assume that evaluating a condition requires constant time and that accessing a register also takes constant time. In the worst case, after a new element/event arrives, we need to evaluate all of the conditions/minterms on the c outgoing transitions of the current state to determine which one of them is triggered. We may also need to access all of the r registers in order to evaluate the conditions. Therefore, the complexity of updating the state of the *dSRT* T is $O(c + r)$ (assuming that each register is accessed only once and its contents are provided to every condition which references that register).

In the general case though, non-deterministic *SRT* are used because they can report exactly the complex events that are detected at every timepoint k and not just the fact that (at least one) such complex event has been detected. The solution to the problem of estimating the runtime complexity of our CER engine when using non-deterministic *SRT* is thus not as straightforward as in the case of deterministic *SRT*.

For our implementation of *SRT*-based CER, we have used Wayeb as a starting point. Wayeb is a Complex Event Recognition and Forecasting engine, based on symbolic automata [9, 12] [1]. We have extended Wayeb so that it can compile (windowed) *SREMO* into *SRT* and then use non-deterministic *SRT* for recognition. We have implemented both the compiler from *SREMO* to *SRT* and the *SRT* as well.

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 (see Section 4.2). Subsequently, Wayeb creates a streaming version of this *SRT* T_s (see Section 4.3). This streaming *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 24), $[1, T'.q_s, \sharp]$. $S_{..0}$ 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 $\varrho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k, v_k]$ 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 $\varrho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k, v_k] \xrightarrow{\delta_k} [k+1, q_{k+1}, v_{k+1}]$, 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 may be updated as usual for one of those next states. For each of the other $n-1$ next states, ϱ is first cloned, producing $n-1$ new runs ϱ', ϱ'', etc. Then each of these runs is updated with the new state and register contents

- $\varrho' = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k, v_k] \xrightarrow{\delta'_k} [k+1, q'_{k+1}, v'_{k+1}]$
- $\varrho'' = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{k-1}} [k, q_k, v_k] \xrightarrow{\delta''_k} [k+1, q''_{k+1}, v''_{k+1}]$
- ...

[1] Available here: `https://github.com/ElAlev/Wayeb`.

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 $Run_f(T_s, S_{..k})$

```
1  Run_f(T_s, S_..k) ← ∅;
2  Run(T_s, S_..k) ← ∅;
3  foreach ϱ ∈ Run(T_s, S_..k−1) do
4      C ← FindSuccessorConfigurations(ϱ, t_k);
5      if |C| > 0 then
6          c ← pick and remove element from C;
7          ϱ_new ← UpdateRun(ϱ, c);
8          if IsAccepting(ϱ_new) then
9              ReportMatch(ϱ_new);
10             Run_f(T_s, S_..k) ← Run_f(T_s, S_..k) ∪ ϱ_new;
11         else
12             Run(T_s, S_..k) ← Run(T_s, S_..k) ∪ ϱ_new;
13         foreach c ∈ C do
14             ϱ' ← Clone(ϱ);
15             ϱ_new ← UpdateRun(ϱ', c);
16             if IsAccepting(ϱ_new) then
17                 ReportMatch(ϱ_new);
18                 Run_f(T_s, S_..k) ← Run_f(T_s, S_..k) ∪ ϱ_new;
19             else
20                 Run(T_s, S_..k) ← Run(T_s, S_..k) ∪ ϱ_new;
21 return Run_f(T_s, S_..k),Run(T_s, S_..k);
```

The updated/new runs are added to the set of runs $Run(T_s, S_{..k})$. Accepting runs are the exception here. 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 the remaining runs of $Run(T_s, S_{..k-1})$.

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 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

$$|Run(T_s, S_{..k-1})| \cdot n_p \cdot c_p + |Run(T_s, S_{..k-1})| \cdot (n_p - 1) \cdot c_c + |Run(T_s, S_{..k-1})| \cdot n_p \cdot c_u$$

or

$$|Run(T_s, S_{..k-1})| \cdot (n_p \cdot c_p + \cdot (n_p - 1) \cdot c_c + n_p \cdot c_u) = |Run(T_s, S_{..k-1})| \cdot (n_p \cdot (c_p + c_c + c_u) - c_c)$$

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 [49]. 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.

Note that in the current version of our engine we have not performed any algorithmic optimizations. Each run is internally in a minimalistic manner, represented by 3-tuple, holding the current state of the run, its register contents and a list with the indices of the simple events it has marked at every timepoint. Thus, we do not need to explicitly represent each run as a separate class instance and we avoid the cost of cloning and maintaining run objects. We have not implemented any postponing ([49]) or match-sharing ([15]) optimizations. We have only implemented some simple code optimizations. Our engine has been written in Scala. Scala generally favors the use of structures such as Sets and Lists and the use of methods such as .map and .filter on such structures. We have avoided the use of such structures and methods in critical parts of the code, since they have proven to be sub-optimal. Instead, we opted for arrays and while loops with indices. This C-like implementation proved to be substantially better in terms of performance.

7 Experimental results

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 (which is the motivation for introducing symbolic register transducers). For this reason, we had to exclude systems without the ability to express relational patterns, such as CORE and 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, as mentioned in [14].

We also considered Flink's implementation of MATCH_RECOGNIZE [3]. 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, without any attempt at parallelization/distribution in order to ensure a level comparison field (note that Esper and FlinkCEP support parallelization). Wayeb is an open-source engine and the experiments presented here are reproducible [2].

As a basis for our experiments, we used the benchmark suite presented in [15] [3]. 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, the total amount, etc.

The suite allows to run the same pattern on multiple engines and with multiple windows. As explained in the previous section, the runtime complexity of a CER engine with a given pattern depends heavily on the size of the window and the number of Kleene operators which the pattern contains. For this reason, we have used the ability of the suite to run experiments on multiple windows. We also focused on patterns with (multiple) Kleene operators. For all patterns, we fixed the selection policy to skip-till-any-match, since this is the most demanding policy, both in terms of time and space complexity.

Our results are presented incrementally as we increase the complexity of the tested

[2] Available here: `https://github.com/ElAlev/cer-srt`.
[3] `https://github.com/CORE-cer/CORE-experiments`.

patterns. We start with sequential patterns where some of the simple events are related through constraints (Section 7.1). We also study the effect of window size on such patterns (Section 7.2). We then add Kleene operators on single events to these patterns (Section 7.3). We additionally test patterns with nested Kleene operators (Section 7.4). Finally, we present results with patterns containing various, mixed operators (Section 7.5). Note that it is not possible to test all systems against all classes of patterns. Some systems either do not support all classes or have ambiguous/different semantics compared to the "expected" ones [27]. In these cases, we thus restrict our comparison to the systems which can actually accommodate the target patterns. For all patterns, we used windows, even though Wayeb can accommodate windowless patterns. 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 "reporting a complex event". Typically, a complex event is "reported" by being printed on the standard output, stored in a file/database or pushed, via a messaging system (such as Kafka), to other "event consumers". However, such steps are generally expensive and system/architecture dependent, which would make throughput estimations less robust. In order to address this issue, we perform a different step after every complex event detection, instead of "reporting" it. We scan every simple event contained in a complex event, we check whether the remainder of the division of an event's timestamp by 10 is 0 and increment a counter if this condition is satisfied. We thus avoid the cost of accessing the standard output, files and/or databases while ensuring that complex events are not "ignored" and undergo a certain, minimal processing.

We considered using implementation-independent metrics in order to "properly" 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 [15, 49]. 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. (**??**) 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].

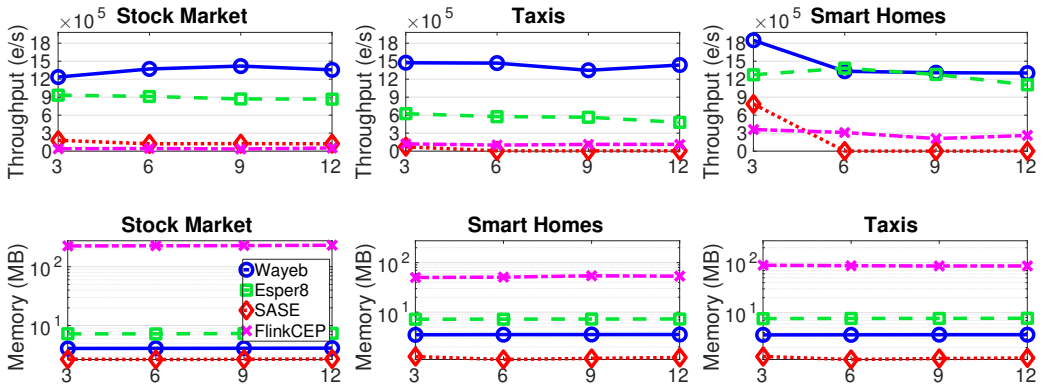


Figure 6 Throughput and memory consumption for sequential patterns with n-ary predicates as a function of pattern length. Window sizes are $w_{stock} = 500$, $w_{smart} = 5$, $w_{taxi} = 100$.

7.1 Sequential patterns

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

$$seq_3 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{5}$$

where $\circlearrowleft$ denotes the skip-till-any-match selection strategy (see Definition 44), w is the window size and ϕ_1, ϕ_2, ϕ_3 all contain "local" constraints, i.e., conditions applied to the single, most recently read event. ϕ_3 also contains a condition relating the most recently read input event with the event that triggered ϕ_1. For example, in the stock market dataset, we have:

$$\begin{aligned} \phi_1(x) &:= x.name = INTC \\ \phi_2(x) &:= x.name = RIMM \\ \phi_3(x, y) &:= (x.name = QQQ \wedge x.price > y.price) \end{aligned}$$

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. (5) 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:

$$\begin{aligned} seq_6 := \circlearrowleft ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim, r_1) \uparrow \bullet), \\ (\phi_4(\sim) \uparrow \bullet \downarrow r_2), (\phi_5(\sim) \uparrow \bullet), (\phi_6(\sim, r_2) \uparrow \bullet))^{[1..w]} \end{aligned} \tag{6}$$

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 sub-expressions we also add a relational constraint (e.g., between ϕ_4 and ϕ_6 in Pattern (6)). 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

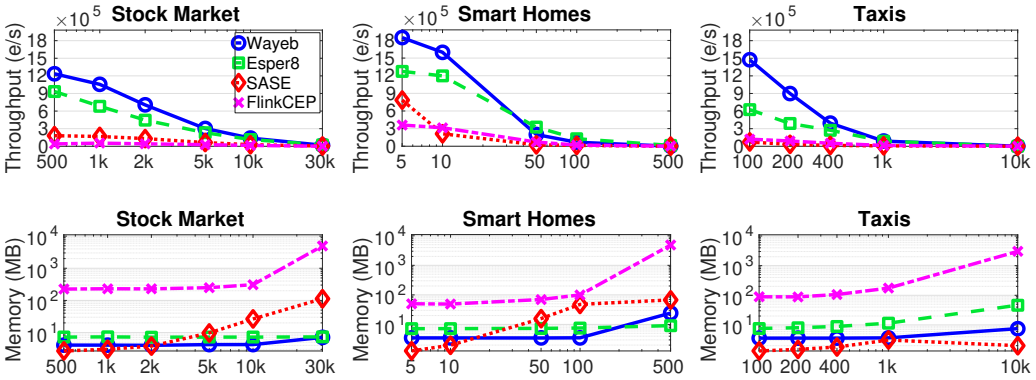


Figure 7 Throughput and memory consumption for sequential patterns with n-ary predicates as a function of window size. Pattern length is 3.

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 6 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. Interestingly, the length of the pattern does not always have a negative effect on throughput. A decrease in throughput is significant only for the smart homes 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. 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.

7.2 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 7 shows the relevant results. Wayeb again 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 presented results.

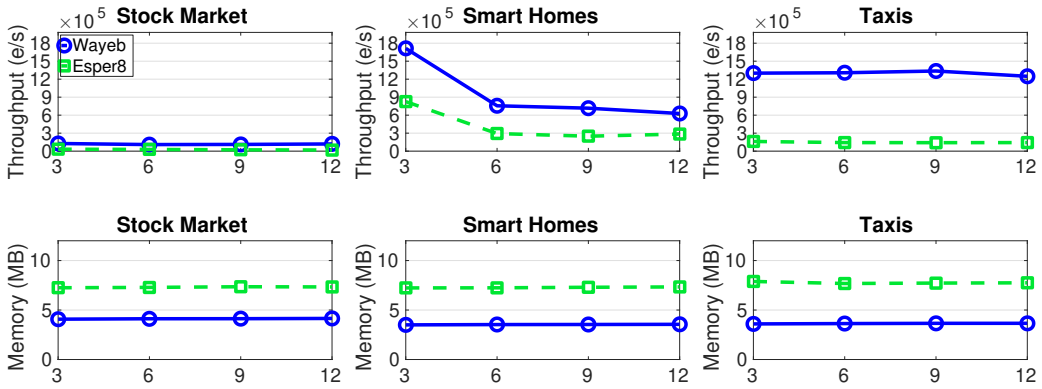


Figure 8 Throughput and memory consumption for patterns with n-ary predicates and Kleene operators as a function of pattern length. SASE and FlinkCEP are excluded because they do not support patterns with Kleene operators with the expected semantics.

7.3 Patterns with Kleene operators

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

$$k_3 := \circlearrowleft\ ((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet)^+, (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{7}$$

Pattern (7) is the same as Pattern (5), with a single difference. $(\phi_2(\sim) \uparrow \bullet)^+$ is a Kleene-plus operation, i.e., standing for $\phi_2(\sim) \cdot (\phi_2(\sim) \uparrow \bullet)^*$, one or more iterations of ϕ_2 ($\phi^+ := \phi \cdot \phi^*$). 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 8 shows the throughput and memory results. 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.

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. Wayeb also has a lower memory footprint. 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.

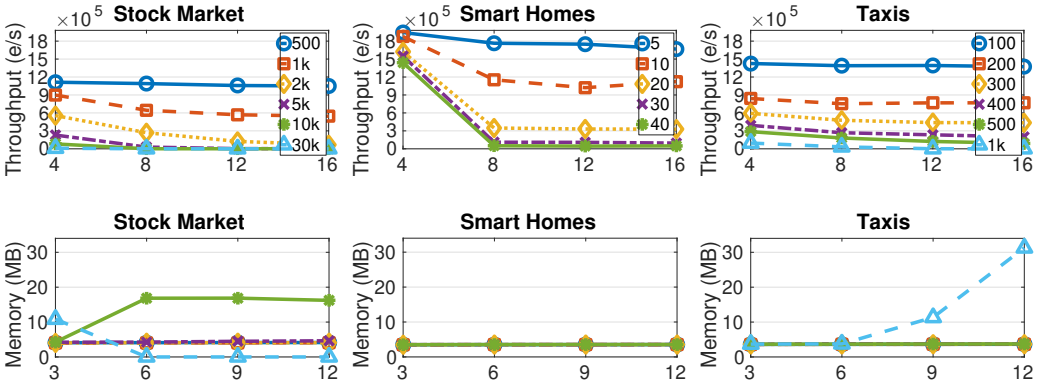


Figure 9 Throughput and memory consumption for patterns with n-ary predicates and nested Kleene operators as a function of pattern length for various windows. SASE, FlinkCEP and Esper are excluded because they do not support patterns with nested Kleene operators.

7.4 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:

$$kn_4 := \circlearrowleft((\phi_1(\sim)\uparrow \bullet \downarrow r_1), ((\phi_2(\sim)\uparrow \bullet), (\phi_3(\sim)\uparrow \bullet)^+)^+, (\phi_4(\sim, r_1)\uparrow \bullet))^{[1..w]} \tag{8}$$

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.

SASE's language does not support patterns with nested Kleene operators. FlinkCEP has the issues mentioned in Section 7.3 regarding the semantics of skip-till-any-match with iteration. Esper's language is also not able to support such patterns. Thus, Wayeb is the only engine which can properly support nested Kleene operators.

The experimental results are shown in Figure 9. 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 as Wayeb consumes a stream. 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.

7.5 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. These operators include simple filters, disjunction and combinations of iteration and disjunction. In particular, we tested 5 patterns:

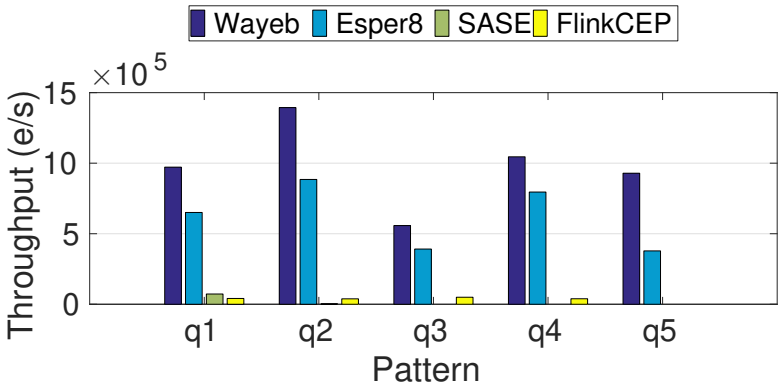


Figure 10 Throughput for patterns with n-ary predicates and various operators. $w = 1000$.

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

$$q_1 := \circlearrowleft((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet), (\phi_3(\sim) \uparrow \bullet), (\phi_4(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{9}$$

where

$$\begin{aligned}\phi_1(x) &:= x.type = SELL \wedge x.name = MSFT\\ \phi_2(x) &:= x.type = BUY \wedge x.name = ORCL\\ \phi_3(x) &:= x.type = BUY \wedge x.name = CSCO\\ \phi_3(x,y) &:= x.type = SELL \wedge x.name = AMAT \wedge x.price < y.price\end{aligned}$$

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: $\phi_2(x) := (x.type = BUY \vee x.type = SELL) \wedge x.name = ORCL$. 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:

$$q_5 := \circlearrowleft((\phi_1(\sim) \uparrow \bullet \downarrow r_1), (\phi_2(\sim) \uparrow \bullet)^+, (\phi_3(\sim, r_1) \uparrow \bullet))^{[1..w]} \tag{10}$$

where

$$\begin{aligned}\phi_2(x) := (&x.type = BUY \vee x.type = SELL) \wedge\\ &x.name = QQQ \wedge x.volume = 4000\end{aligned}$$

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 relevant results are shown in Figure 10. 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. This is 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 modified Wayeb's behavior in two ways. First, we disallowed any post-processing/reporting of the detected complex events. Second, 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. In both cases, 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. This result also indicates that the main bottleneck for Wayeb lies in the actual evaluation and maintenance of the various runs and not in the production of their output.

8 Discussion and Conclusions

We presented a system for CER based on an automaton model, *SRT*, that can act as a computational model for patterns with n-ary conditions ($n \geq 1$), which are quintessential for CER applications. *SRT* have nice compositional properties, without imposing severe restrictions on the use of operators. Most of the standard operators in CER, such as concatenation/sequence, union/disjunction, intersection/conjunction and Kleene-star/iteration, may be used freely. We showed that complement may also be used and determinization is possible, if a window operator is used, a very common feature in CER. We briefly discussed the complexity of the problems of non-emptiness, membership and universality. Although the problem of membership in general is at least NP-complete, in cases where we can use windowed, deterministic *SRT*, the cost of updating the state of such an automaton after reading a single element is linear in the number of registers and conditions. With non-deterministic *SRT*, the runtime complexity becomes exponential in the size of the window and the number of Kleene-star operators. We presented experimental results showing that our framework with *SRT* is highly expressive, with the ability to support complex patterns with nested operators and relational constraints. For instance, it is the only system that may express in practice nested Kleene operators. At the same time, we do not need to sacrifice performance for this increased expressive power. It also outperforms other state-of-the-art engines for most patterns and workloads.

It is interesting that our system can achieve this even without any algorithmic optimizations. Our aim for the future is to investigate our engine's optimization potential. For example, CORE exploits structural and computational commonalities in order to speed up the processing of matches [15]. It employs a graph structure to represent all matches compactly and to avoid redundant predicate evaluations. However, CORE's patterns carry only unary conditions, i.e., relational constraints are not allowed. This means that some graph-based optimization techniques cannot be directly transferred to *SRT*. Other optimization techniques include lazy evaluation of runs [24] and various distribution methods (see [27] for an overview). We also aim to extend our framework towards Complex Event Forecasting. This could potentially allow us to investigate optimization techniques based on "branch prediction", i.e. based on predictions on how a run might evolve in the future.

8.1 Variable binding and aggregates

Another line of importance concerns the ability of *SREMO* and *SRT* to capture aggregates, which is related to how the current semantics of *SREMO* determine (register) variable binding.

Table 4 Example stream.

type	B	B	B	S	B	S	...
id	1	1	1	1	1	1	...
index	1	2	3	4	5	6	...

As an example, consider the following *SREMO*:

$$e := (\mathit{TypeIsB}(\sim) \uparrow \bullet \downarrow r_1)^+ \cdot ((\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \uparrow \bullet) \tag{11}$$

One question is the following: how exactly is variable r_1 mapped to input events? The sub-expression inside the Kleene-plus operator requires the storage of one or more B events in register r_1, which will be compared later for identifier equality with a S event. According to the semantics of *SREMO*, if there are multiple B events before the S event, register r_1 will be overwritten. When the S event arrives, it will be compared only with the last (stored) B event.

On the other hand, it is often useful and more intuitive to expect different semantics: for the pattern to be satisfied, all B events must be compared to the S event. This type of semantics would also be useful for aggregates. For example, instead of equality, we might require that the sum of the volumes of all B events exceeds a given threshold. Essentially, in both cases, we would require each B event to be stored in a different register, something which is not currently allowed by the semantics of *SREMO*.

In order to support aggregates, the first step would be to modify the semantics of *SREMO* so as to have a stack of registers where an iteration operator can store events. If *SRT* are restricted to windowed *SREMO* (aggregates for unbounded or windowless iteration are in any case not very meaningful), then *SRT* could support aggregates. This could be achieved by "unrolling" iterations (similarly to the way we unroll them for determinization, see technical report). This method would be computationally sub-optimal, since it would potentially require a large number of registers for storing multiple events inside an iteration operator, but it is sufficient to show that *SRT* are expressive enough to capture aggregates of single, non-nested iterations. In practice, appropriate optimizations would need to be employed, which we intend to explore in the future.

For nested iterations, we would probably need to move beyond *SRT*. Consider the following *SREMO*:

$$e' := ((\mathit{TypeIsB}(\sim) \uparrow \bullet \downarrow r_1)^+ \cdot ((\mathit{TypeIsS}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \uparrow \bullet))^+$$

It is the same as *SREMO* (11), but enclosed in an extra Kleene-star operator. Assume we apply this pattern on the stream of Table 4. When we reach the sixth event in the stream ($\mathit{index} = 6$), should we compare its id with the id of the fifth event only, or with the ids of events 1, 2 and 3 as well? If the intended semantics is that of the former option (i.e., the comparison should not include all B events, but only those of the "current" repetition), then we would probably need to extend *SRT*, possibly with special flags indicating which registers belong to which repetition of an iteration operator. We intend to explore in the future more precisely the necessary modifications of *SRT* required for handling more flexible variable binding schemes and multiple semantics for aggregates.

8.2 Hierarchies of complex events

Finally, we comment briefly on the issue of if and how *SREMO* and *SRT* may be used to construct complex event hierarchies, i.e., automata which can process not only simple input events but the output of other automata as well. Defining complex events in terms of other complex events is feasible at the language level, given the compositionality of *SREMO* operators. Each instance of a complex event definition e within another definition e' could be replaced by the initial definition of e and then compile e' into a single *SRT*. However, this would be a sub-optimal solution in cases where e appears in multiple other definitions, since the sub-automaton corresponding to e would be constructed multiple times and each new input event would need to be processed repeatedly by all these copies. This solution might not even be possible in a distributed CER setting where the results of a sub-automaton need to be sent to another automaton in a different location which does not have access to the original input events. Constructing hierarchies properly is non-trivial and raises several issues concerning the semantics of operators, e.g., the associativity of sequence [48]. Hence, although we have not currently implemented a mechanism for constructing hierarchies, it is worth noting that: a) our language is compositional, a property which paves the way for a proper treatment of hierarchies, and b) our system currently produces complex events as sets of indices, i.e., the complete history of a match. As shown in [48], this is the only model with the necessary properties to avoid all temporal issues with hierarchies. In the future, we will investigate how we can extend *SRT* so that they can handle complete histories as input (presumably at the upper levels of a hierarchy).

A Appendix

A.1 Proof of Theorem 20

▷ Theorem. Let e, e' be two *SREMO*. If, for every string S, $Match(e, S) = Match(e', S)$, then $Lang(e) = Lang(e')$.

Proof. We need to show that $\forall S, S \in Lang(e) \Leftrightarrow S \in Lang(e')$. First, assume that $S \in Lang(e)$. We also know that $Match(e, S) = Match(e', S)$. Thus, from the definition of matches (see Definition 18), it follows that $(e, S, M, \sharp) \vdash v$ for some M and some v. It also holds that $(e', S, M, \sharp) \vdash v'$. Then, by definition (see again Definition 18), $S \in Lang(e')$. We have thus proven that $S \in Lang(e) \Rightarrow S \in Lang(e')$. With a similar reasoning, we can also prove that $S \in Lang(e') \Rightarrow S \in Lang(e)$. ◀

A.2 Proof of Theorem 27

▷ Theorem. 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.

Proof. We only need to prove that $Match(e, S) = Match(T, S)$ for every string S. $Lang(e) = Lang(T)$ then follows immediately from Theorem 20.

For a *SREMO* e, a string S and valuations v, v', let $\mathcal{M}(e, S, v, v')$ denote all matches M such that $(e, S, M, v) \vdash v'$. Similarly, for a *SRT* T, let $\mathcal{M}(T, S, v, v')$ denote all matches M such that $M \in Match(\varrho)$ where $\varrho \in Run_f(T, S)$ and $\varrho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} [n, q_{n+1}, v_{n+1}]$, with $v_1 = v$ and $v_{n+1} = v'$. For every possible *SREMO* e, we will construct a corresponding *SRT* T and then prove either that $Match(e, S) = Match(T, S)$ or that $\mathcal{M}(e, S, v, v') = \mathcal{M}(T, S, v, v')$ for every string S. The latter implies that $\mathcal{M}(e, S, \sharp, v'') =$

$\mathcal{M}(T, S, \sharp, v'')$ for some valuation v'' or equivalently $Match(e, S) = Match(T, S)$, which is our goal. The proof is inductive. We prove directly the base cases for the simple expressions $e := \emptyset$, $e := \epsilon$, $e := \phi = R(x_1, \cdots, x_n)$ and $e := \phi = R(x_1, \cdots, x_n) \downarrow w$. For the complex expression $e := e_1 \cdot e_2$, $e := e_1 + e_2$ and $e' = e^*$, we use as an inductive hypothesis that our target result hods for the sub-expressions and then prove that it also holds for the top expression. For example, for $e := e_1 \cdot e_2$, we assume that $\mathcal{M}(e_1, S_1, v, v'') = \mathcal{M}(T_1, S_1, v, v'')$ and that $\mathcal{M}(e_2, S_2, v'', v') = \mathcal{M}(T_2, S_2, v'', v')$.

We must be careful, however, with the valuations. If, for example, v applies to the *SRT* T, does it also apply to the sub-automaton T_1, if T and T_1 have different registers? We can avoid this problem and make all valuations compatible (i.e., having the same domain as functions) by fixing the registers for all expressions and sub-expressions. We can estimate the registers that we need for a top expression e by scanning its conditions and write operations. Let $reg(e)$ be a function applied to a *SREM* e. We define it as follows:

$$reg(e) = \begin{cases} \emptyset & \text{if } e = \emptyset \\ \emptyset & \text{if } e = \epsilon \\ \{x_1\} \cup \cdots \cup \{x_n\} \cup \{w\} & \text{if } e = R(x_1, \cdots, x_n) \downarrow w \\ reg(e_1) \cup reg(e_2) & \text{if } e = e_1 \cdot e_2 \\ reg(e_1) \cup reg(e_2) & \text{if } e = e_1 + e_2 \\ reg(e_1) & \text{if } e = (e_1)^* \end{cases} \tag{12}$$

For our proofs that follow, we first apply this function to the top expression e to obtain $R_{top} = reg(e)$ and we use R_{top} as the set of registers for all automata and sub-automata. All valuations can thus be compared without any difficulties, since they will have the same domain R_{top}.

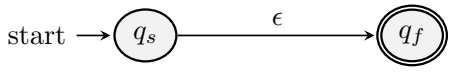
Assume $e := \epsilon$. We know that $Match(e, S) = \emptyset$. We can then construct a *SRT* $T = (Q, q_s, Q_f, R, \Delta)$ where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \epsilon \uparrow \otimes \downarrow \emptyset \rightarrow q_f$. See Figure 11a. It is obvious that T accepts only the empty string since there is only one path that leads to the final state and this path goes through an ϵ transition. No elements are marked. Thus $Match(T, S) = \emptyset$.

Assume $e := \phi = R(x_1, \cdots, x_n) \uparrow o$, where ϕ is a condition and all x_i belong to a set of register variables $\{r_1, \cdots, r_k\}$. We construct the following *SRT* $T = (Q, q_s, Q_f, R, \Delta)$, where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \phi \uparrow o \downarrow \emptyset \rightarrow q_f$. See Figure 11b.

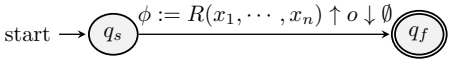
We first prove $M \in \mathcal{M}(e, S, v, v') \Rightarrow M \in \mathcal{M}(T, S, v, v')$ for a match M and any string S. We also first assume that $o = \bullet$. It is obvious that S must be composed of a single element, i.e., $S = t_1$. Also, $M = \{1\}$. Since $S = t_1$ is accepted by e starting from the valuation v, this means that $(\phi, S, M, v) \vdash v'$, with $v' = v$, according to the second case of Definition 17. Thus $(t_1, v) \models \phi$. This then implies that the second case in the definition of a successor configuration (see Definition 23) holds for our constructed automaton T. As a result, T, upon reading S, moves to its final state q_f, marks t_1 and accepts S. This move does not change the valuation, thus $v' = v$. We have thus proven that $M = \{1\} \in \mathcal{M}(T, S, v, v')$. Similarly, if $o = \otimes$, we can prove that $M = \emptyset \in \mathcal{M}(T, S, v, v')$.

The inverse direction, $M \in \mathcal{M}(T, S, v, v') \Rightarrow M \in \mathcal{M}(e, S, v, v')$, can be proven in a similar manner.

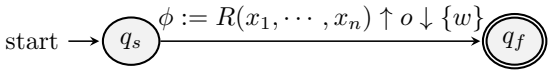
Assume $e := \phi = R(x_1, \cdots, x_n) \uparrow o \downarrow w$, where ϕ is a condition, all x_i belong to a set of register variables $\{r_1, \cdots, r_k\}$ and w a write register (not necessarily one of r_i). We construct the following *SRT* $T = (Q, q_s, Q_f, R, \Delta)$, where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$,



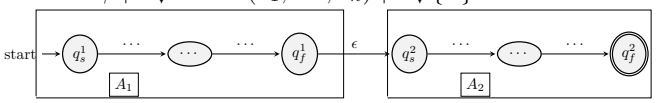
(a) Base case of a single ϵ condition, $e := \epsilon$.



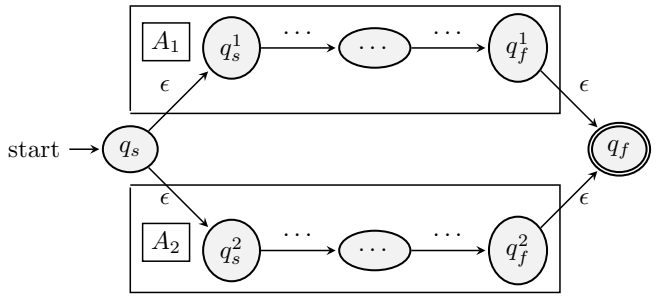
(b) Base case of a single condition, $e := \phi \uparrow o = R(x_1, \cdots, x_n) \uparrow o$.



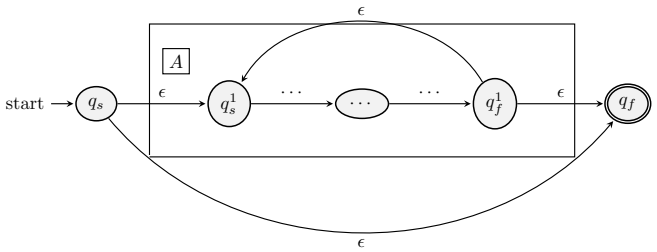
(c) Base case of a single condition with a write register, $e := \phi \uparrow o \downarrow W = R(x_1, \cdots, x_n) \uparrow o \downarrow \{w\}$.



(d) Concatenation. $e = e_1 \cdot e_2$.



(e) OR. $e = e_1 + e_2$.



(f) Iteration. $e^{'} = e^*$.

Figure 11 The cases for constructing a *SRT* from a *SREMO*.

$R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \phi \uparrow o \downarrow \{w\} \rightarrow q_f$. See Figure 11c.

The proof is essentially the same as that for the previous case. The only difference is that we need to use the third case from the definition of successor configurations (Definition 23). This means that $v' = v[w \leftarrow t_1]$. If $w \in R$, then t_1 is stored in w and $v'(w) = t_1$. Otherwise, v' remains the same as v.

Assume $e := e_1 \cdot e_2$, where e_1 and e_2 are *SREMO*. We first construct T_1 and T_2, the *SRT* for e_1 and e_2 respectively. We construct the following *SRT* $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = T_1.Q \cup T_2.Q$, $q_s = T_1.q_s$, $Q_f = \{T_2.q_f\}$, $R = R_{top}$, $\Delta = T_1.\Delta \cup T_2.\Delta \cup \{\delta\}$ and $\delta = T_1.q_f, \epsilon \rightarrow T_2.q_s$. See Figure 11d. We thus simply connect T_1 and T_2 with an ϵ transition. Notice that $T_1.R$ and $T_2.R$ may overlap. Their union retains only one copy of each register, if a register appears in both of them.

We first prove $M \in \mathcal{M}(e, S, v, v') \Rightarrow M \in \mathcal{M}(T, S, v, v')$ for a match M and any string S. Since $M \in \mathcal{M}(e, S, v, v')$, S can be broken into two sub-strings S_1 and S_2 and M into two subsets M_1 and M_2 such that $S = S_1 \cdot S_2$, $M = M_1 \cdot M_2$, $(e_1, S_1, M_1, v) \vdash v''$ and $(e_2, S_2, M_2, v'') \vdash v'$. This is equivalent to $M_1 \in \mathcal{M}(e_1, S_1, v, v'')$ and $M_2 \in \mathcal{L}(e_2, S_2, v'', v')$. From the induction hypothesis (i.e., that what we want to prove holds for the sub-expressions e_1, e_2 and their automata T_1, T_2) it follows that $M_1 \in \mathcal{M}(T_1, S_1, v, v'')$ and $M_2 \in \mathcal{M}(T_2, S_2, v'', v')$. Notice that if T_1 and t_2 have different sets of registers, we can always expand $T_1.R$ and $T_2.R$ to their union, without affecting in any way the behavior of the automata. Now, let $l_1 = |S_1|$ and $l_2 = |S_2|$. From $M_1 \in \mathcal{M}(T_1, S_1, v, v'')$ it follows that there exists an accepting run ϱ_1 of T_1 over S_1 such that $\varrho_1 = [1, T_1.q_s, v] \rightarrow \cdots \rightarrow [l_1 + 1, T_1.q_f, v'']$. Similarly, from $M_2 \in \mathcal{M}(T_2, S_2, v'', v')$ it follows that there exists an accepting run ϱ_2 of T_2 over S_2 such that $\varrho_2 = [1, T_2.q_s, v''] \rightarrow \cdots \rightarrow [l_2 + 1, T_2.q_f, v']$. Let's construct a run by connecting ϱ_1 and ϱ_2 with an ϵ transition: $\varrho = [1, T_1.q_s, v] \rightarrow \cdots \rightarrow [l_1 + 1, T_1.q_f, v''] \overset{T_1.q_f, \epsilon \rightarrow T_2.q_s}{\rightarrow} [l_1 + 2, T_2.q_s, v''] \rightarrow \cdots \rightarrow [l_1 + l_2 + 1, T_2.q_f, v']$. We can see that this is indeed an accepting run of T and that $Match(\varrho) = M$. Thus $M \in \mathcal{M}(T, S, v, v')$.

The inverse direction, $M \in \mathcal{M}(T, S, v, v') \Rightarrow M \in \mathcal{M}(e, S, v, v')$, can be proven in a similar manner. Since $M \in \mathcal{M}(T, S, v, v')$, there exists an accepting run ϱ of T over S. By the construction of T, however, this run must be in the form $\varrho = \varrho_1 \overset{\epsilon}{\rightarrow} \varrho_2$, with ϱ_1 being an accepting run of T_1 over a string S_1 and ϱ_2 an accepting run of T_2 over S_2, where $S = S_1 \cdot S_2$. We then use the induction hypothesis to prove that $M_1 = Match(\varrho_1) \in \mathcal{M}(e_1, S_1, v, v'')$ and $M_2 = Match(\varrho_2) \in \mathcal{L}(e_2, S_2, v'', v')$ and finally that $M = M_1 \cdot M_2 \in \mathcal{M}(e, S, v, v')$.

Assume $e := e_1 + e_2$, where e_1 and e_2 are *SREMO*. We first construct T_1 and T_2, the *SRT* for e_1 and e_2 respectively. We construct the following *SRT* $T = (Q, q_s, Q_f, R, \Delta)$, where $Q = T_1.Q \cup T_2.Q \cup \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = T_1.\Delta \cup T_2.\Delta \cup \{\delta_{s,1}, \delta_{s,2}, \delta_{1,f}, \delta_{2,f}\}$ and $\delta_{s,1} = q_s, \epsilon \rightarrow T_1.q_s$, $\delta_{s,2} = q_s, \epsilon \rightarrow T_2.q_s$, $\delta_{1,f} = T_1.q_f, \epsilon \rightarrow q_f$, $\delta_{2,f} = T_2.q_f, \epsilon \rightarrow q_f$. See Figure 11e. We thus create a new state, q_s, acting as the start state and connect it through ϵ transitions to the start states of T_1 and T_2. We also create a new final state and connect to it the final states of T_1 and T_2. Again, $T_1.R$ and $T_2.R$ may overlap. Their union retains only one copy of each register, if a register appears in both of them.

It is easy to prove that $M \in \mathcal{M}(e, S, v, v') \Rightarrow M \in \mathcal{M}(T, S, v, v')$ for a match M and any string S. If $(e_1, S, M, v) \vdash v'$, this implies that e_1 is accepted by T_1 and M is a match of e_1 and T_1. M is thus also a match of T. Similarly if $(e_2, S, M, v) \vdash v'$ for T_2. The inverse direction has a similar proof.

Assume $e' := e^*$, where e is a *SREM*. We construct a new *SRA* T' as shown in Figure 11f. We first construct the *SRA* for e, T. We create a new final and a new start state. We connect the new start state to the old start and to the new final. We connect the old final to the new final and the old start. R is again R_{top}.

We first prove that $M \in \mathcal{M}(e', S, v, v') \Rightarrow M \in \mathcal{M}(T', S, v, v')$ for a match M and any string S. Since $M \in \mathcal{M}(e', S, v, v')$, $S = S_1 \cdot S'$ and $M = M_1 \cdot M'$ such that $(e, S_1, M_1, v) \vdash v''$ and $(e^*, S', M', v'') \vdash v'$. Equivalently, this implies that

- $(e, S_1, M_1, v) \vdash v_1$ and $M_1 = Match(\varrho_1) \in \mathcal{M}(T, S_1, v, v_1)$
- $(e, S_2, M_2, v_1) \vdash v_2$ and $M_2 = Match(\varrho_2) \in \mathcal{M}(T, S_2, v_1, v_2)$
- $(e, S_3, M_3, v_2) \vdash v_3$ and $M_3 = Match(\varrho_3) \in \mathcal{M}(T, S_3, v_2, v_3)$ etc
- until $(e, S_n, M_n, v_{n-1}) \vdash v_n$ and $M_n = Match(\varrho_n) \in \mathcal{M}(T, S_n, v_{n-1}, v_n)$,

where $v_n = v'$. We can then construct the run $\varrho = \varrho_1 \xrightarrow{\epsilon} \varrho_2 \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \varrho_n$. It is easy to see that ϱ is an accepting run of T' and that $M = M_1 \cdot M_2 \cdot \cdots \cdot M_n$ is a match of T'. Similarly for the inverse direction. ◀

A.3 Proof of Lemma 29

▷ Lemma. For every *SRT* T_ϵ with ϵ transitions there exists an equivalent *SRT* $T_{\notin}$ without ϵ transitions, i.e., a *SRT* such that $Match(T_\epsilon, S) = Match(T_{\notin}, S)$ for every string S.

ALGORITHM 2: Eliminating ϵ-transitions (*EliminateEpsilon*).

Input: *SRT* T_ϵ, possibly with ϵ transitions
Output: *SRT* $T_{\notin}$ without ϵ-transitions

$q_{\notin,s} \leftarrow Enclose(T_\epsilon.q_s)$; $Q_{\notin} \leftarrow \{q_{\notin,s}\}$; $\Delta_{\notin} \leftarrow \emptyset$;
if $\exists q \in q_{\notin,s} : q \in T_\epsilon.Q_f$ **then**
  $Q_{\notin,f} \leftarrow \{q_{\notin,s}\}$;
else
  $Q_{\notin,f} \leftarrow \emptyset$;
frontier $\leftarrow \{q_{\notin,s}\}$;
while *frontier* $\neq \emptyset$ **do**
  $q_{\notin} \leftarrow$ pick an element from *frontier*;
  $t \leftarrow$ gather all outgoing transitions (except epsilon) of all $q_\epsilon \in q_{\notin}$;
  cos $\leftarrow$ find all distinct pairs of conditions and outputs from t;
  foreach *co* $\in$ *cos* **do**
    $W \leftarrow$ gather all write registers from all t whose condition and output match that of *co*;
    $p_{\notin} \leftarrow$ gather and enclose all target states from all t whose condition and output match that of *co*;
    $Q_{\notin} \leftarrow Q_{\notin} \cup \{p_{\notin}\}$;
    if $\exists q \in p_{\notin} : q \in T_\epsilon.Q_f$ **then**
      $Q_{\notin,f} \leftarrow Q_{\notin,f} \cup \{p_{\notin}\}$;
    $\delta_{\notin} \leftarrow CreateNewTransition(q_{\notin}, co.\phi \uparrow co.o \downarrow W \rightarrow p_{\notin})$;
    $\Delta_{\notin} \leftarrow \Delta_{\notin} \cup \{\delta_{\notin}\}$;
    frontier $\leftarrow$ *frontier* $\cup \{p_{\notin}\}$;
  frontier $\leftarrow$ *frontier* $\setminus \{q_{\notin}\}$;
$T_{\notin} \leftarrow (Q_{\notin}, q_{\notin,s}, Q_{\notin,f}, T_\epsilon.R, \Delta_{\notin})$;
`return` $T_{\notin}$;

Proof. We first give the algorithm. See Algorithm 2. Note that in this algorithm, the function *Enclose* is the usual function for ϵ-enclosure in standard automata theory and we

will not repeat it here (see [33]). Suffice it to say that, when applied to a state q (or set of states $\{q_i\}$), it returns all the states we can reach from q (or all q_i) by following only ϵ-transitions. It is also worth noting that the algorithm does not create the power-set of states and then connects them through transitions. It creates those subsets it needs by "forward-looking" for what is necessary, but it is equivalent to the power-set construction algorithm. We will prove that $Match(T_\epsilon, S) = Match(T_{\notin}, S)$ for every string S or, equivalently, that $M \in \mathcal{M}(T_\epsilon, S, \sharp, v') \Leftrightarrow M \in \mathcal{M}(T_{\notin}, S, \sharp, v')$ for a match M and any string S.

We first prove the direction $M \in \mathcal{M}(T_\epsilon, S, v, v') \Rightarrow M \in \mathcal{M}(T_{\notin}, S, v, v')$. The other direction can be proven similarly. Let ϱ_ϵ denote an accepting run of A_ϵ over S, where $k = |S|$ is the length of S.

$$
\begin{aligned}
\varrho_\epsilon = & [1, q_{\epsilon,1} = q_{\epsilon,s}, v_{\epsilon,1} = \sharp] \xrightarrow{\epsilon} [\cdots] \xrightarrow{\epsilon} \cdots && \text{sub-run 1} \\
& \xrightarrow{\delta_{\epsilon,1}} [2, q_{\epsilon,2}, v_{\epsilon,2}] \xrightarrow{\epsilon} [\cdots] \xrightarrow{\epsilon} \cdots && \text{sub-run 2} \\
& \cdots \\
& \xrightarrow{\delta_{\epsilon,i-1}} [i, q_{\epsilon,i}, v_{\epsilon,i}] \xrightarrow{\epsilon} [\cdots] \xrightarrow{\epsilon} [i', q_{\epsilon,i'}, v_{\epsilon,i'}] && \text{sub-run i} \\
& \xrightarrow{\delta_{\epsilon,i}} [i+1, q_{\epsilon,i+1}, v_{\epsilon,i+1}] \xrightarrow{\epsilon} [\cdots] \xrightarrow{\epsilon} \cdots && \text{sub-run i+1} \\
& \cdots \\
& \xrightarrow{\delta_{\epsilon,k}} [k+1, q_{\epsilon,k+1} \in Q_{\epsilon,f}, v_{\epsilon,k+1}] && \text{sub-run k+1}
\end{aligned}
\tag{13}
$$

Let $\varrho_{\notin}$ denote a run of $A_{\notin}$ over S.

$$
\begin{aligned}
\varrho_{\notin} = & [1, q_{\notin,1} = q_{\notin,s}, v_{\notin,1} = \sharp] && \text{sub-run 1} \\
& \xrightarrow{\delta_{\notin,1}} [2, q_{\notin,2}, v_{\notin,2}] && \text{sub-run 2} \\
& \cdots \\
& \xrightarrow{\delta_{\notin,i-1}} [i, q_{\notin,i}, v_{\notin,i}] && \text{sub-run i} \\
& \xrightarrow{\delta_{\notin,i}} [i+1, q_{\notin,i+1}, v_{\notin,i+1}] && \text{sub-run i+1} \\
& \cdots \\
& \xrightarrow{\delta_{\notin,k}} [k+1, q_{\notin,k+1}, v_{\notin,k+1}] && \text{sub-run k+1}
\end{aligned}
\tag{14}
$$

$\varrho^{\notin}$ necessarily follows k transitions, since it does not have any ϵ-transitions. On the other hand, ϱ^{ϵ} may follow more than k transitions ($j \geq k$), because several ϵ transitions may intervene between "actual", non-ϵ transitions, as shown in Run (13). The number of non-ϵ transitions is still k. ϱ_ϵ is thus necessarily composed of k "sub-runs", where the first configuration of each sub-run is reached via a non-ϵ transition, followed by a sequence of 0 or more ϵ transitions. Each line in Run (13) is such a sub-run. We can also split $\varrho_{\notin}$ in sub-runs, but in this case each such sub-run will be simply composed of a single configuration. See Run (14).

We will prove the following. For each sub-run i of ϱ_ϵ, it holds that:

1. $q_{\epsilon,i} \in q_{\notin,i}$.
2. $v_{\epsilon,i} = v_{\notin,i}$, i.e., T_ϵ and $T_{\notin}$ have the same register contents at each i.
3. $\delta_{\epsilon,i}.o = \delta_{\notin,i}.o$.

We can prove this inductively. We assume that the above claims hold for i and then we can show that they must necessarily hold for $i+1$. Since they are obviously true for $i = 1$, they are then true for all other i as well. Thus, $q_{\notin,k+1} \in Q_{\notin,f}$, $\varrho_{\notin}$ is an accepting run as well and $Match(\varrho_\epsilon) = Match(\varrho_{\notin})$.

First, notice that within each sub-run i of ϱ_ϵ, $v_{\epsilon,i}$ remains the same, since ϵ transitions never modify the contents of the registers. Thus, in ϱ_ϵ, $v_{\epsilon,i'} = v_{\epsilon,i}$. It is also obviously true that $i' = i$, since ϵ transitions do not read elements from S and thus the automaton's head does not move. The only thing that could possibly change is $q_{\epsilon,i'}$, so that, in general, $q_{\epsilon,i'} \neq q_{\epsilon,i}$. Therefore, in Run (13), we move from sub-run i to sub-run $i+1$ by jumping from $q_{\epsilon,i'}$ to $q_{\epsilon,i+1}$. This implies that $\delta_{\epsilon,i}$, connecting $q_{\epsilon,i'}$ to $q_{\epsilon,i+1}$, is triggered when the contents of the register are those of $v_{\epsilon,i'} = v_{\epsilon,i}$.

Now, $q_{\epsilon,i'}$ belongs to the enclosure of $q_{\epsilon,i}$. Otherwise, it would be impossible to reach it from $q_{\epsilon,i}$ by following only ϵ transitions. From the induction hypothesis we know that $q_{\epsilon,i} \in q_{\notin,i}$. From the construction algorithm for $T_\notin$ (Algorithm 2) we also know that the transition $\delta_{\epsilon,i}$ also exists in $T_\notin$, with $q_{\notin,i}$ as its source. $\delta_{\notin,i}$ has the same condition/output and references the same registers as $\delta_{\epsilon,i}$. Since $\delta_{\epsilon,i}$ is triggered with $v_{\epsilon,i'}$, $\delta_{\notin,i}$ must also be triggered because $v_{\notin,i} = v_{\epsilon,i}$ (by the induction hypothesis) and thus $v_{\notin,i} = v_{\epsilon,i'}$. From the construction algorithm, we can see that $q_{\epsilon,i+1} \in q_{\notin,i+1}$. The state $q_\notin$ in Algorithm 2 is $q_{\notin,i}$ in Run (14) whereas $p_\notin$ is $q_{\notin,i+1}$. $p_\notin$ contains all states that can be reached from $q_{\epsilon,i}$ when $\delta_{\epsilon,i}$ is triggered. Thus, it also contains $q_{\epsilon,i+1}$.

The second part of the induction hypothesis is obviously true for $i+1$, i.e., $v_{\epsilon,i+1} = v_{\notin,i+1}$, since exactly the same registers are modified in exactly the same way by $\delta_{\epsilon,i}$ and $\delta_{\notin,i}$.

The third part is also true. The construction algorithm ensures that $\delta_{\epsilon,i}.o = \delta_{\notin,i}.o$.

Therefore, $q_{\epsilon,k+1} \in q_{\notin,k+1}$ which implies that $q_{\notin,k+1} \in Q_{\notin,f}$ and thus $\varrho_\notin$ is an accepting run of $T_\notin$ over S. Moreover, the transitions between sub-runs in Runs (13) and (14) have marked exactly the same input elements. Therefore, if $M \in \mathcal{M}(T_\epsilon, S, \sharp, v')$, then $M \in \mathcal{M}(T_\notin, S, \sharp, v')$. ◀

A.4 Proof of Theorem 31

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

Proof. For union, concatenation and Kleene-star the proof is essentially the proof for converting *SREMO* to *SRT*. For concatenation, if we have *SRT* T_1 and T_2 we construct T as in Figure 11d. For union, we construct the *SRA* as in Figure 11e. For Kleene-star, we construct the *SRA* as in Figure 11f. The only difference in these constructions is that we now assume, without loss of generality, that $T_1.R \cap T_2.R = \emptyset$, i.e., that T_1 and T_2 have different sets of registers and that the automaton T constructed from T_1 and T_2 retains all registers of both T_1 and T_2. For example, if we have two *SRT* T_1 and T_2 and we want to construct a *SRT* T such that $Match(T, S) = Match(T_1, S) \cdot Match(T_2, S)$ then we connect T_1's final state to T_2's start state via an ϵ transition. It is easy to see that if $M_1 \in Match(T_1, S)$ and $M_2 \in Match(T_2, S)$ then $M = M_1 \cdot M_2 \in Match(T, S)$. S_1 will force T to move to $T_1's$ final state (both A and T_1 start with empty registers). Subsequently, T will jump to T_2's start state and then S_2 will force T to go to T_2's final state which is T's final state, since T_2's registers in T are empty when T_2 starts reading S_2.

We will now prove closure under intersection. Let $T_1 = (Q_1, q_{1,s}, Q_{1,f}, R_1, \Delta_1)$ and $T_2 = (Q_2, q_{2,s}, Q_{2,f}, R_2, \Delta_2)$ be two *SRT*. We wan to construct a *SRT* $T = (Q, q_s, Q_f, R, \Delta)$ such that $Match(T, S) = Match(T_1, S) \cap Match(T_2, S)$. We construct T as follows:

- $Q = Q_1 \times Q_2$.
- $q_s = (q_{1,s}, q_{2,s})$.
- $Q_f = (q_1, q_2)$, where $q_1 \in Q_{1,f}$ and $q_2 \in Q_{2,f}$, i.e., $Q_f = Q_{1,f} \times Q_{2,f}$.
- $R = R_1 \cup R_2$, assuming, without loss of generality, that $R_1 \cap R_2 = \emptyset$.

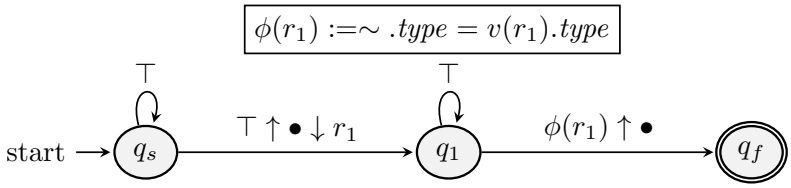


Figure 12 *SRT* accepting strings which have the same type in two elements. Notice that $\sim$ denotes the current event (last event read from the string).

- For each $q = (q_1, q_2) \in Q$ we add a transition δ to $q' = (q_1', q_2') \in Q$ if there exists a transition δ_1 from q_1 to q_1' in T_1 and a transition δ_2 from q_2 to q_2' in T_2. The condition of δ is $\phi = \delta_1.\phi \wedge \delta_2.\phi$. The output of δ is $\bullet$ if the outputs of δ_1 and δ_2 are both $\bullet$. Otherwise, it is $\otimes$. The write registers of δ are $W = \delta_1.W \cup \delta_2.W$ (notice that, if $\delta_1.W \neq \emptyset$ and $\delta_2.W \neq \emptyset$, this creates a multi-register *SRT*, even if T_1 and T_2 are single-register). Thus, $\delta = (q_1, q_2), (\delta_1.\phi \wedge \delta_2.\phi) \downarrow (\delta_1.W \cup \delta_2.W) \rightarrow (q_1', q_2')$.

It is evident that, if a match M is produced by both T_1 and T_2 on a string S, it is also produced by T. If M is not produced either by T_1 or T_2, then it is not produced by T. Therefore, $Match(T, S) = Match(T_1, S) \cap Match(T_2, S)$.

◀

A.5 Proof of Theorem 35

▷ Theorem. *SRT* are not closed under complement.

Proof. The proof is by a counter example. Let T denote the *SRT* of Figure 12. This *SRT* reads strings composed of tuples. Each tuple contains an attribute called *type*, taking values from a finite or infinite alphabet. The symbol $\sim$ simply denotes the current element of the string, i.e., the last element read from it. Therefore, T accepts strings in which there are two elements with the same type, regardless of the length of S. Assume that there exists a *SRT* T_c which accepts only when T does not accept. In other words, T_c accepts all strings S whose elements all have a different type. Let $k = |T_c.R|$ be the number of registers of T_c. Let $|S| = k + m$, where $m > 1$, be the length of a string S whose elements all have different types. However, T_c cannot possibly exist. At the end of S, as T_c is ready to read the last element of S, it must have stored all of the previous $k + m - 1$ elements of S. But T has only k registers, whereas $k + m - 1 > k$, since $m > 1$. Thus, T_c cannot exist. ◀

A.6 Proof of Theorem 38

▷ Theorem. Not every *SRT* is output-agnostic determinizable.

Proof. The proof is again by a counter example. Let T denote the *SRT* of Figure 13. This *SRT* reads strings composed of tuples. Each tuple contains an attribute, called *type*, taking values from a finite or infinite alphabet. It also contains another tuple, called *id*, taking integer values. T thus accepts strings S that contain an a followed by a b, whose ids are equal, regardless of the length of S.

Assume there exist a deterministic *SRT* T_d with k registers which is equivalent to T. Let

$S = (a, 1)(b, 2)$

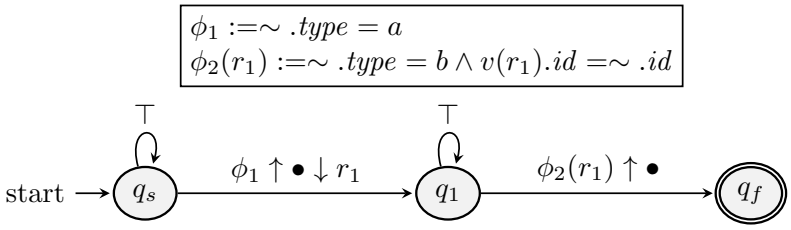


Figure 13 *SRT* accepting all strings containing an a element followed by a b element, whose identifiers are the same.

be a string given to T_d. After reading $S_1 = (a, 1)$, A_d must store it in a register r_1 in order to be able to compare it when $(b, 2)$ arrives. Let

$$S' = (a, 1)(a, 3)(b, 2)$$

After reading $S'_1 = (a, 1)$, T_d must store it in the register r_1, since T_d is deterministic and follows a single run. Thus, it must have the exact same behavior after reading S_1 and S'_1. But we must also store $S'_2 = (a, 3)$ after reading it. Additionally, S'_2 must be stored in a different register r_2. We cannot overwrite r_1. If we did this and S'_1 were $(a, 2)$, then we would not be able to match $(a, 2)$ to $S'_3 = (b, 2)$ and $S' = (a, 2)(a, 3)(b, 2)$ would not be accepted. Now, let

$$S'' = \underbrace{(a, \cdots)(a, \cdots)\cdots(a, \cdots)}_{k+1 \text{ elements}}(b, 2)$$

With a similar reasoning, all of the first $k + 1$ elements of S'' must be stored after reading them. But this is a contradiction, as T_d can store at most k different elements. Therefore, there does not exist a deterministic *SRT* which is equivalent to T. ◀

A.7 Proof of Theorem 41

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

Proof. In what follows, we omit referring to the output of transitions, since we will be focusing on output-agnostic determinism. We will thus treat *SRT* as if they were automata without output. We call such automata *Symbolic Register Automata* (*SRA*) (very similar to Symbolic Register Automata presented in [19]). Equivalence between deterministic and non-deterministic *SRA* will be shown at the level of languages, not that of matches.

We first show how we can construct a so-called "unrolled *SRA*" from a windowed expression:

▶ **Lemma 46.** *For every windowed SREMO there exists an equivalent unrolled SRA without any loops, i.e., a SRA where each state may be visited at most once.*

Proof. Let $e_w := e^{[1..w]}$. Algorithm 3 shows how we can construct A_{e_w}. The basic idea is that we first construct as usual the *SRA* A_e for the sub-expression e (and eliminate ϵ-transitions). We can then use A_e to enumerate all the possible walks of A_e of length up to w and then join them in a single *SRA* through disjunction. A walk w over a *SRA* A is a sequence of transitions $w =< \delta_1, \cdots, \delta_k >$, such that:

- $\forall \delta_i \ \delta_i \in A.\Delta$

ALGORITHM 3: Constructing *SRA* for a windowed expression (*ConstructWSRA*).

Input: Windowed *SREMO* $e' := e^{[1..w]}$
Output: *SRA* $A_{e'}$ equivalent to e'

$A_{e,\epsilon} \leftarrow ConstructSRA(e)$; `// As described in Appendix A.2.`
$A_{e,ms} \leftarrow EliminateEpsilon(A_{e,\epsilon})$; `// See Algorithm 2.` $A_{e,ms}$ `might be multi-register.`
$A_e \leftarrow ConvertToSingleRegister(A_{e,ms})$; `// As described in [11].`
$A_{e'} \leftarrow Unroll(A_e, w)$; `// See Algorithm 4.`
`return` $A_{e'}$;

ALGORITHM 4: Unrolling cycles for windowed expressions (*Unroll*).

Input: *SRA* A and integer $k \geq 0$
Output: *SRA* A_k with runs of length up to k

if $k = 0$ **then**
    $(A_k, Frontier, CopyOfQ, CopyOfR) \leftarrow Unroll0(A)$; `// Algorithm 5`
else
    $(A_k, Frontier, CopyOfQ, CopyOfR) \leftarrow UnrollK(A, k)$; `// Algorithm 6`
end
`return` $(A_k, Frontier, CopyOfQ, CopyOfR)$;

ALGORITHM 5: Unrolling cycles for windowed expressions, base case (*Unroll0*).

Input: *SRA* A
Output: *SRA* A_0 with runs of length 0

$q \leftarrow CreateNewState()$;
$CopyOfQ \leftarrow \{q \rightarrow A.q_s\}$;
$CopyOfR \leftarrow \emptyset$;
$Frontier \leftarrow \{q\}$;
$Q_f \leftarrow \emptyset$;
if $A.q_s \in A.Q_f$ **then**
    $Q_f \leftarrow Q_f \cup \{q\}$;
end
$A_0 \leftarrow (\{q\}, q, Q_f, \emptyset, \emptyset)$;
`return` $(A_0, Frontier, CopyOfQ, CopyOfR)$;

ALGORITHM 6: Unrolling cycles for windowed expressions, $k > 0$ (*UnrollK*).

Input: *SRA* A and integer $k > 0$
Output: *SRA* A_k with runs of length up to k

$(A_{k-1}, Frontier, CopyOfQ, CopyOfR) \leftarrow Unroll(A, k-1)$;
$NextFrontier \leftarrow \emptyset$;
$Q_k \leftarrow A_{k-1}.Q$; $Q_{k,f} \leftarrow A_{k-1}.Q_f$; $R_k \leftarrow A_{k-1}.R$; $\Delta_k \leftarrow A_{k-1}.\Delta$;
foreach $q \in Frontier$ **do**
  $q_c \leftarrow CopyOfQ(q)$;
  foreach $\delta \in A.\Delta : \delta.source = q_c$ **do**
    $q_{new} \leftarrow CreateNewState()$;
    $Q_k \leftarrow Q_k \cup \{q_{new}\}$;
    $CopyOfQ \leftarrow CopyOfQ \cup \{q_{new} \rightarrow \delta.target\}$;
    if $\delta.target \in A.Q_f$ **then**
      $Q_{k,f} \leftarrow Q_{k,f} \cup \{q_{new}\}$;
    if $\delta.W = \emptyset$ **then**
      $R_{new} \leftarrow \emptyset$;
    else
      $r_{new} \leftarrow CreateNewRegister()$;
      $R_k \leftarrow R_k \cup \{r_{new}\}$;
      $R_{new} \leftarrow \{r_{new}\}$;
      $CopyOfR \leftarrow CopyOfR \cup \{r_{new} \rightarrow \delta.r\}$; // $\delta.r$ single element of $\delta.W$
    $\phi_{new} \leftarrow \delta.\phi$;
    $rs_{new} \leftarrow ()$;
    /* By $\delta.\phi.rs$ we denote the register selection of $\delta.\phi$, i.e., all the registers referenced by $\delta.\phi$ in its arguments. rs is represented as a list. */
    foreach $r \in \delta.\phi.rs$ **do**
      /* *FindLastAppearance* returns a register that is a copy of r and appears last in the trail of A_{k-1} to q (no other copies of r appear after r_{latest}). Due to the construction, only a single walk/trail to q exists. */
      $r_{latest} \leftarrow FindLastAppearance(r, q, A_{k-1})$;
      /* :: denotes the operation of appending an element at the end of a list. r_{latest} is appended at the end of rs_{new}. */
      $rs_{new} \leftarrow rs_{new} :: r_{latest}$;
    $\delta_{new} \leftarrow CreateNewTransition(q, \phi_{new}(rs_{new}) \downarrow R_{new} \rightarrow q_{new})$;
    $\Delta_k \leftarrow \Delta_k \cup \{\delta_{new}\}$;
    $NextFrontier \leftarrow NextFrontier \cup \{q_{new}\}$;
$A_k \leftarrow (Q_k, A_{k-1}.q_s, Q_{k,f}, R_k, \Delta_k)$;
return $(A_k, NextFrontier, CopyOfQ, CopyOfR)$;

- $\delta_1.source = A.q_s$
- $\forall \delta_i, \delta_{i+1}\ \delta_i.target = \delta_{i+1}.source$

We say that such a walk is of length k. Essentially, we need to remove cycles from every walk of A_e by "unrolling" them as many times as necessary, without the length of the walk exceeding w. This "unrolling" operation is performed by the (recursive) Algorithm 4. Because of this "unrolling", a state of A_e may appear multiple times as a state in A_{e_w}. We keep track of which states of A_{e_w} correspond to states of A_e through the function *CopyOfQ* in the algorithm. For example, if q_e is a state of A_e, q_{e_w} a state of A_{e_w} and $CopyOfQ(q_{e_w}) = q_e$, this means that q_{e_w} was created as a copy of q_e (and multiple states of A_{e_w} may be copies of the same state of A_e, i.e., *CopyOfQ* is a surjective but not an injective function). We do the same for the registers as well, through the function *CopyOfR*. The algorithm avoids an explicit enumeration, by gradually building the automaton as needed, through an incremental expansion. Of course, walks that do not end in a final state may be removed, either after the construction or online, whenever a non-final state cannot be expanded.

The lemma is a direct consequence of the construction algorithm. First, note that, by the construction algorithm, there is a one-to-one mapping (bijective function) between the walks/runs of A_{e_w} and the walks/runs of A_e of length up to w. We can show that if ϱ_e is a run of A_e of length up to w over a string S (ϱ_e has at most w transitions), then the corresponding run ϱ_{e_w} of A_{e_w} is indeed a run and if ϱ_e is accepting so is ϱ_{e_w}. By definition, since the runs have no ϵ-transitions and are at most of length w, $|S| \leq w$.

We first prove the following proposition:

▷ Proposition. There exists a run of A_e over a string S of length up to w

$$\varrho_e = [1, q_{e,1} = A_e.q_s, v_{e,1}] \overset{\delta_{e,1}}{\rightarrow} \cdots \overset{\delta_{e,i-1}}{\rightarrow} [n, q_{e,i}, v_{e,i}] \overset{\delta_{e,i}}{\rightarrow} \cdots \overset{\delta_{e,n-1}}{\rightarrow} [n, q_{e,n}, v_{e,n}]$$

iff there exists a run ϱ_{e_w} of A_{e_w}

$$\varrho_{e_w} = [1, q_{e_w,1} = A_{e_w}.q_s, v_{e_w,1}] \overset{\delta_{e_w,1}}{\rightarrow} \cdots \overset{\delta_{e_w,i-1}}{\rightarrow} [n, q_{e_w,i}, v_{e_w,i}] \overset{\delta_{e_w,i}}{\rightarrow} \cdots \overset{\delta_{e_w,n-1}}{\rightarrow} [n, q_{e_w,n}, v_{e_w,n}]$$

such that:

- $CopyOfQ(q_{e_w,i}) = q_{e,i}$
- $v_{e,i}(r_e) = v_{e_w,i}(r_{e_w})$, if $CopyOfR(r_{e_w}) = r_e$ and r_{e_w} appears last among the registers that are copies of r_e in ϱ_{e_w}.

We say that a register r appears in a run at position i if $r \in \delta_i.W$, i.e., if the i^{th} transition writes to r. We say that a register r_{e_w}, where $CopyOfR(r_{e_w}) = r_e$, appears last if no other copies of r_e appear after r_{e_w} in a run. The notion of a register's (last) appearance also applies for walks of A_{e_w}, since A_{e_w} is a directed acyclic graph, as can be seen by Algorithms 5 and 6 (they always expand "forward" the *SRA*, without creating any cycles and without merging any paths).

Proof. The proof is by induction on the length of the runs k, with $k \leq w$. We prove only one direction (assume a run ϱ_e exists). The other is similar.

Base case: $k = 0$. For both *SRA*, only the start state and the initial configuration with all registers empty is possible. Thus, $v_{e,i} = v_{e_w,i} = \sharp$ for all registers. By Algorithm 5 (line 2), we know that $CopyOf(q_{e_w,s}) = q_{e,s}$.

Case for $0 < k + 1 \leq w$, assuming the proposition holds for k. Let

$$\varrho_{e,k+1} = \cdots [k, q_{e,k}, v_{e,k}] \overset{\delta_{e,k}}{\rightarrow} [k+1, q_{e,k+1}, v_{e,k+1}]$$

and

$$\varrho_{e_w,k+1} = \cdots [k, q_{e_w,k}, v_{e_w,k}] \overset{\delta_{e_w,k}}{\rightarrow} [k+1, q_{e_w,k+1}, v_{e_w,k+1}]$$

be the runs of A_e and A_{e_w} respectively of length $k+1$ over the same $k+1$ elements of a string S. We know that $\varrho_{e,k+1}$ is an actual run and we need to construct $\varrho_{e_w,k+1}$, knowing, by the induction hypothesis, that there is an actual run up to $q_{e_w,i+k}$. Now, by the construction algorithm, we can see that if $\delta_{e,k}$ is a transition of A_e from $q_{e,k}$ to $q_{e,k+1}$, there exists a transition $\delta_{e_w,k}$ with the same condition from $q_{e_w,k}$ to a $q_{e_w,k+1}$ such that $CopyOfQ(q_{e_w,k+1}) = q_{e,k+1}$. Moreover, if $\delta_{e,k}$ is triggered, so does $\delta_{e_w,k}$, because the registers in the register selection of $\delta_{e_w,k}$ are copies of the corresponding registers in $\delta_{e,k}.\phi.rs$. By the induction hypothesis, we know that the contents of the registers in $\delta_{e,k}.\phi.rs$ will be equal to the contents of their corresponding registers in ϱ_{e_w} that appear last. But these are exactly the registers in $\delta_{e_w,k}.\phi.rs$ (see line 22 in Algorithm 6). We can also see that the part of the proposition concerning the valuations v also holds. If $\delta_{e,k}.W = \{r_e\}$ and $\delta_{e_w,k}.W = \{r_{e_w}\}$, then we know, by the construction algorithm (line 18), that $CopyOfR(r_{e_w}) = r_e$ and r_{e_w} will be the last appearance of a copy of r_e in $\varrho_{e_w,k+1}$. Thus the proposition holds for $0 < k+1 \leq w$ as well. ◀

The above proposition must necessarily hold for accepting runs as well. Therefore, A_e accepts the same language as A_{e_w}. ◀

We also note that w must be a number greater than (or equal to) the minimum length of the walks induced by the accepting runs of A_e (which is something that can be computed by the structure of the expression). Although this is not a formal requirement, if it is not satisfied, then A_{e_w} won't detect any matches.

The process for constructing a deterministic *SRA* (*dSRA*) from a windowed *SREMO* is shown in Algorithm 7. It first constructs a non-deterministic *SRA* (*nSRA*) and then uses the power set of this *nSRA*'s states to construct the *dSRA*. For each state q_d of the *dSRA*, it gathers all the conditions from the outgoing transitions of the states of the *nSRA* q_n ($q_n \in q_d$), it creates the (mutually exclusive) *minterms* of these conditions, i.e., the set of maximal satisfiable Boolean combinations of the conditions. It then creates transitions, based on these minterms. Please, note that we use the ability of a transition to write to more than one registers. So, from now on, $\delta.W$ will be a set that is not necessarily a singleton. This allows us to retain the same set of registers, i.e., the set of registers R will be the same for the *nSRA* and the *dSRA*. A new transition created for the *dSRA* may write to multiple registers, if it "encodes" multiple transitions of the *nSRA*, which may write to different registers. It is also obvious that the resulting *SRA* is deterministic, since the various minterms out of every state are mutually exclusive, i.e., at most one may be triggered. Intuitively, having a windowed *SRA* allows us to construct a deterministic *SRA* with as many registers as necessary. Therefore, it is always possible to have available all past w elements. This is not possible in the counter-example of Section A.6, where we showed that *SRA* are not in general determinizable.

First, we will prove the following proposition:

▷ Proposition. There exists a run ϱ_n over a string S which A_n can follow by reading the first k tuples of S, iff there exists a run ϱ_d that A_d can follow by reading the same first k tuples, such that, if

$$\varrho_n = [1, q_{n,1}, v_{n,1}] \overset{\delta_{n,1}}{\rightarrow} \cdots \overset{\delta_{n,i-1}}{\rightarrow} [i, q_{n,k}, v_{n,i}] \overset{\delta_{n,i}}{\rightarrow} \cdots \overset{\delta_{n,k-1}}{\rightarrow} [k, q_{n,k}, v_{n,k}]$$

and

$$\varrho_d = [1, q_{d,1}, v_{d,1}] \overset{\delta_{d,1}}{\rightarrow} \cdots \overset{\delta_{d,i-1}}{\rightarrow} [i, q_{d,i}, v_{d,i}] \overset{\delta_{d,i}}{\rightarrow} \cdots \overset{\delta_{d,k-1}}{\rightarrow} [k, q_{d,k}, v_{d,k}]$$

are the runs of A_n and A_d respectively, then,

- $q_{n,i} \in q_{d,i}\ \forall i : 1 \leq i \leq k$
- if $r \in A_d.R$ appears in ϱ_n, then it appears in ϱ_d
- $v_{n,i}(r) = v_{d,i}(r)$ for every r that appears in ϱ_n (and ϱ_d)

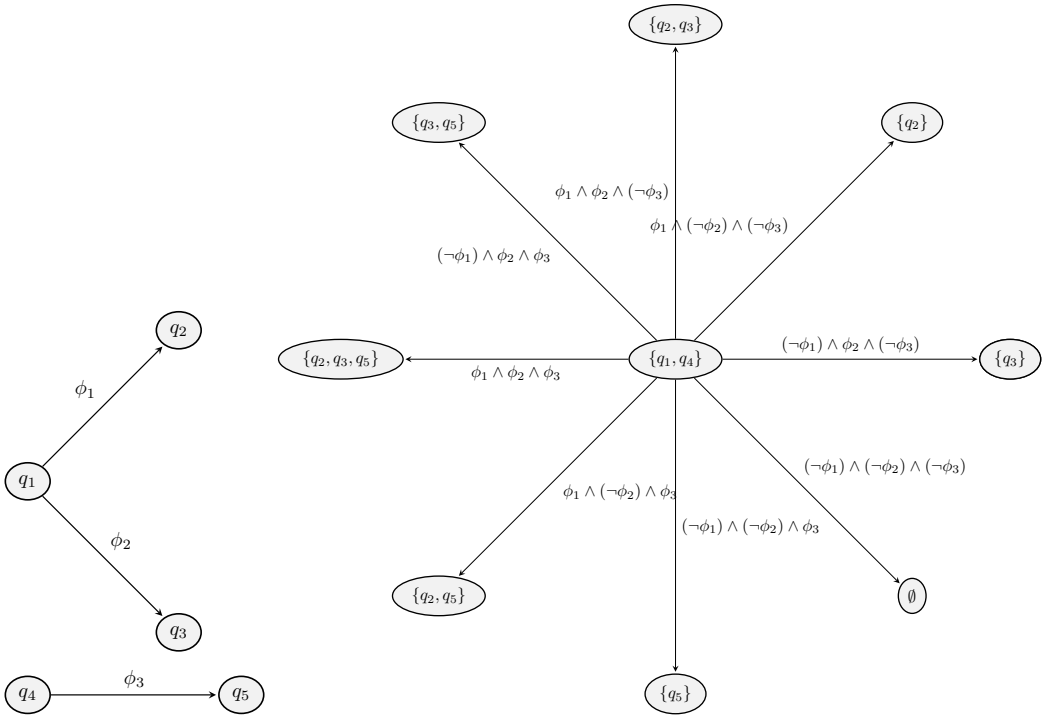
We say that a register r appears in a run at position i if $r \in \delta_i.W$.

ALGORITHM 7: Determinization.

Input: Windowed *SREMO* $e' := e^{[1..n]}$
Output: Deterministic *SRA* A_d equivalent to e'

$A_n \leftarrow ConstructWSRA(e')$; `// See Algorithm 3`
$Q_d \leftarrow ConstructPowerSet(A_n.Q)$;
$\Delta_d \leftarrow \emptyset$; $Q_{f,d} \leftarrow \emptyset$;
foreach $q_d \in Q_d$ **do**
  if $q_d \cap A_n.Q_f \neq \emptyset$ **then**
    $Q_{f,d} \leftarrow Q_{f,d} \cup \{q_d\}$;
  $Conditions \leftarrow ()$; $rs_d \leftarrow ()$;
  foreach $q_n \in q_d$ **do**
    foreach $\delta_n \in A_n.\Delta : \delta_n.source = q_n$ **do**
      $Conditions \leftarrow Conditions :: \delta_n.\phi$;
      $rs_d \leftarrow rs_d :: \delta_n.\phi.rs$;
  `/*` *ConstructMinTerms* `returns the min-terms from a set of conditions. For example, if` $Conditions = (\phi_1, \phi_2)$`, then` $MinTerms = (\phi_1 \wedge \phi_2, \neg\phi_1 \wedge \phi_2, \phi_1 \wedge \neg\phi_2, \neg\phi_1 \wedge \neg\phi_2)$ `*/`
  $MinTerms \leftarrow ConstructMinTerms(Conditions)$;
  foreach $mt \in MinTerms$ **do**
    $p_d \leftarrow \emptyset$; $W_d \leftarrow \emptyset$;
    foreach $q_n \in q_d$ **do**
      foreach $\delta_n \in A_n.\Delta : \delta_n.source = q_n$ **do**
        `/*` $\phi \vDash \psi$ `denotes entailment, i.e., if` ϕ `is true then` ψ `is necessarily also true. For example,` $\phi_1 \wedge \neg\phi_2 \vDash \phi_1$. `*/`
        if $mt \vDash \delta_n.\phi$ **then**
          $p_d \leftarrow p_d \cup \{\delta_n.target\}$;
          $W_d \leftarrow W_d \cup \{\delta_n.W\}$;
    $\delta_d \leftarrow CreateNewTransition(q_d, mt(rs_d) \downarrow W_d \rightarrow p_d)$;
    $\Delta_d \leftarrow \Delta_d \cup \{\delta_d\}$;
$q_{d,s} \leftarrow \{A_n.q_s\}$;
$A_d \leftarrow (Q_d, q_{d,s}, Q_{f,d}, A_N.R, \Delta_d)$;
`return` A_d;

Proof. We will prove only direction (the other is similar). Assume there exists a run ϱ_n. We will prove that there exists a run ϱ_d by induction on the length k of the run.



(a) *nSRA*. **(b)** *dSRA*.

Figure 14 Example of converting a *nSRA* to a *dSRA*.

Base case: $k = 0$**.** Then $\varrho_n = [1, q_{n,1}, \sharp] = [1, q_{n,s}, \sharp]$. The run $\varrho_d = [1, q_{d,s}, \sharp]$ is indeed a run of the *dSRA* that satisfies the proposition, since $q_{n,s} \in q_{d,s} = \{q_{n,s}\}$ (by the construction algorithm, line 22), all registers are empty and no registers appear in the runs.

Case $k > 0$**.** Assume the proposition holds for k. We will prove it holds for $k+1$ as well. Let

$$\varrho_{n,k+1} = \cdots [k, q_{n,k}, v_{n,k}] \begin{cases} \overset{\delta^1_{n,k}}{\rightarrow} [k+1, q^1_{n,k+1}, v^1_{n,k+1}] \\ \overset{\delta^2_{n,k}}{\rightarrow} [k+1, q^2_{n,k+1}, v^2_{n,k+1}] \\ \cdots \\ \overset{\delta^m_{n,k}}{\rightarrow} [k+1, q^m_{n,k+1}, v^m_{n,k+1}] \end{cases} \tag{15}$$

be the possible runs that can follow a run $\varrho_{n,k}$ after the *nSRA* reads the $(k+1)^{th}$ tuple. Notice that, typically, since A_n is non-deterministic, there might be multiple runs $\varrho_{n,k}$ and each such run can spawn its own multiple runs $\varrho_{n,k+1}$. The same reasoning that we present below applies to all these $\varrho_{n,k}$.

We need to find a run of the *dSRA* like:

$$\varrho_{d,k+1} = \cdots [k, q_{d,k}, v_{d,k}] \overset{\delta_{d,k}}{\rightarrow} [k+1, q_{d,k+1}, v_{d,k+1}]$$

By the induction hypothesis, we know that $q_{n,k} \in q_{d,k}$. By the construction Algorithm 7, we then know that, if $\phi^j_{n,k} = \delta^j_{n,k}.\phi$ is the condition of a transition that takes the non-deterministic run to $q^j_{n,k+1}$, then there exists a transition $\delta_{d,k}$ in the *dSRA* from $q_{d,k}$ whose condition will be a minterm, containing all the $\phi_{n,k}$ in their positive form and all other

possible conditions in their negated form. Moreover, the target of that transition, $q_{d,k+1}$, contains all $q^j_{n,k+1}$. More formally, $q_{d,k+1} = \bigcup_{j=1}^{m} q^j_{n,k+1}$.

As an example, see Figure 14. Figure 14a depicts part of a *nSRA*. Figure 14b depicts part of the *dSRA* that woyld be constructed from that of Figure 14a. The construction algorithm would create the state $\{q_2, q_4\}$, the minterms from the conditions of all the outgoing transitions of q_2 and q_4 and then attempt to determine which minterm would move the *dSRA* to which subset of $\{q_2, q_3, q_5\}$. The results is shown in Figure 14b. Now, assume that a run of the *nSRA* has reached q_1 via one run and q_4 via another run, i.e. $q_{n,k} = q_1$ in Eq. (15) for the first of these runs and $q_{n,k} = q_4$ for the second. Assume also that both ϕ_1 and ϕ_2 are triggered after reading the $(k+1)^{th}$ element, but not ϕ_3. This means that the *nSRA* would move to q_2 and q_3. In Eq. (15), this would mean that $m = 2$ and that $\delta^1_{n,k}.\phi = \phi_1$ and $\delta^2_{n,k}.\phi = \phi_2$. But in the *dSRA* there is a transition that simulates this move of the *nSRA*. The minterm $\phi_1 \wedge \phi_2 \wedge (\neg\phi_3)$ moves the *dSRA* to $\{q_2, q_3\}$. It contains $\delta^1_{n,k}.\phi$ and $\delta^2_{n,k}.\phi$ in their positive form and all other conditions (here only ϕ_3) in their negated form. With a similar reasoning, we see that the *dSRA* can simulate the *nSRA* for every other possible combination of $\{\phi_1, \phi_2, \phi_3\}$.

What we have proven thus far is a structural similarity between *nSRA* and *dSRA*. We also need to prove that $\delta_{d,k}$ applies as well, i.e., that the minterm on this transition is triggered exactly when its positive conjuncts are triggered. To prove this, we need to show that the contents of the registers that a condition ϕ of the *nSRA* accesses are the same that this ϕ accesses in the *dSRA* when participating in a minterm.

As we said, the condition on $\delta_{d,k}$ is a conjunct (minterm), where all $\phi^j_{n,k}$ appear in their positive form and all other conditions in their negated form. But note that the conditions in negated form are those that were not triggered in $\varrho_{n,k+1}$ when reading the $(k+1)^{th}$ tuple. Additionally, the arguments passed to each of the conditions of the minterm are the same (registers) as those passed to them in the non-deterministic run (by the construction algorithm, line 11). To make this point clearer, consider the following simple example of a minterm:

$$\phi = \phi_1(r_{1,1}, \cdots, r_{1,k}) \wedge \neg\phi_2(r_{2,1}, \cdots, r_{2,l}) \wedge \phi_3(r_{3,1}, \cdots, r_{3,m})$$

This means that $\phi_1(r_{1,1}, \cdots, r_{1,k})$, with the exact same registers as arguments, will be the formula of a transition of the *nSRA*that was triggered. Similarly for ϕ_3. With respect to ϕ_2, it will be the condition of a transition that was not triggered. If we can show that the contents of those registers are the same in the runs of the *nSRA* and *dSRA* when reading the last tuple, then this will mean that $\delta_{d,k}.\phi$ is indeed triggered. But this is the case by the induction hypothesis ($v_{n,k}(r) = v_{d,k}(r)$), since all these registers appear in the run $\varrho_{n,k}$ up to $q_{n,k}$.

The second part of the proposition also holds, since, by the construction, $\delta_{d,k}$ will write to all the registers that the various $\delta^j_{n,k}$ write (see line 19 in the determinization algorithm).

The third part also holds. This is the part that actually ensures that the contents of the registers are the same. First, note that a register can appear only once in a run of A_n, because of its tree-like structure. Second, by the construction, we know that $\delta_{d,k}.W = \bigcup_{j=1}^{m} \delta^j_{n,k}.W$ (see again line 19 in the algorithm). Therefore, we know that $\delta_{d,k}$ will write only to registers that had not appeared before in the run of the *nSRA* and will leave every other register that had appeared unaffected. This observation is critical. We could not claim the same for non-windowed *SRA*, as in Figure 13. If we attempted to determinize this *nSRA*, without unrolling its cycles, the resulting *SRA* could overwrite r_1. Now, since $\delta_{d,k}$ and all the $\delta^j_{n,k}$

write the same element and $\delta_{d,k}$ does not affect any previously appearing registers, the proposition holds. ◀

Since the above proposition holds for accepting runs as well, we can conclude that there exists an accepting run of A_n iff there exists an accepting run of A_d. According to the above proposition, the union of the last states over all ϱ_n is equal to the last state of ϱ_d. Thus, if ϱ_n reaches a final state, then the last state of ϱ_d will contain this final state and hence be itself a final state. Conversely, if ϱ_d reaches a final state of A_d, it means that this state contains a final state of A_n. Then, there must exist a ϱ_n that reached this final state.

◀

A.8 Proof of Corollary 43

▷ Corollary. Windowed *SRT* with ignored outputs are closed under complement.

Proof. Since we ignore outputs, we will be focusing again on *SRA*. Let A be a windowed *SRA*. We first determinize it to obtain A_d. Although A_d is deterministic, it might still be incomplete, i.e., there might be states from which it might be impossible to move to another state. This may happen if it is possible that the conditions on all of the outgoing transitions of such a state are not triggered. As in classical automata, such a behavior implies that the string provided to the automaton is not accepted by it.

We can make A_d complete by adding a so-called "dead" state q_{dead} (non-final) to A_d. See Algorithm 8. For each state q of A_d, we then gather all the conditions on its outgoing transitions. Let Φ denote this set of conditions. We can then create the conjunction of all the negated conditions in Φ: $\phi_{dead} := (\neg\phi_1) \wedge (\neg\phi_2) \wedge \cdots \wedge (\neg\phi_n)$, where $\phi_i \in \Phi$ and $\bigcup_{i=1}^{n} \phi_i = \Phi$. We then add a transition from q to q_{dead} with ϕ_{dead} as its condition and $\emptyset$ as its write registers. If we do this for every state $q \in A_d.Q$, we will have created a *SRA* that is equivalent to A_d, since transitions to q_{dead} are only triggered if none of the other conditions in Φ are triggered. If there exists a condition ϕ_i that is triggered, the new automaton will behave exactly as A_d and if no ϕ is triggered it will go to q_{dead}. Now, if we add a self-loop transition on q_{dead} with $\top$ as its condition, we also ensure that the new automaton will always stay in q_{dead} once it enters it. q_{dead} thus acts as a sink state. This new automaton $A_{d,c}$ will therefore be equivalent to A_d and it will also be both deterministic and complete.

The final move is to flip all the states of $A_{d,c}$, i.e., make all of its final states non-final and all of its non-final states final, to obtain an automaton $A_{complement}$. This then ensures that if a string S is accepted by A (or A_d), it will not be accepted by $A_{complement}$ and if it is accepted by $A_{complement}$ it will not be accepted by A. This is indeed possible because A (and $A_{complement}$) is deterministic and complete. Therefore, for every string S, there exists exactly one run of A (and $A_{complement}$) over S. If A, after reading S, reaches a final state, $A_{complement}$ necessarily reaches a non-final state and vice versa. Therefore, for every windowed *SRA* A we can indeed construct a *SRA* which accepts the complement of the language of A.

Notice that this trick of flipping the states would not be possible if A were non-deterministic. To see this, assume that A is non-deterministic and at the end of S it reaches states q_1 and q_2, where q_1 is non-final and q_2 is final. This means that S is accepted by A. If we flip the states of the non-deterministic A to get its complement $A_{complement}$, we would again reach q_1 and q_2, where, in this case, q_1 is final and q_2 is non-final. $A_{complement}$ would thus again accept S, which is not the desired behavior for $A_{complement}$. ◀

ALGORITHM 8: Constructing the complement of a *SRA* (*Complement*).

Input: Windowed *SRA* A
Output: *SRA* $A_{complement}$ accepting the complement of A's language

$A_d \leftarrow Determinize(A)$; `// See Algorithm 7.`
$q_{dead} \leftarrow CreateNewState()$;
$\Delta_{dead} \leftarrow \emptyset$;
foreach $q \in A_d.Q$ **do**
  $\Phi \leftarrow \emptyset$;
  foreach $\delta \in A_d.\Delta : \delta.source = q$ **do**
    $\Phi \leftarrow \Phi \cup \delta.\phi$;
  end
  $\phi_{dead} \leftarrow \top$;
  foreach $\phi_i \in \Phi$ **do**
    $\phi_{dead} \leftarrow \phi_{dead} \wedge (\neg\phi_i)$;
  end
  $\delta_{dead} \leftarrow CreateNewTransition(q, \phi_{dead} \downarrow \emptyset \rightarrow q_{dead})$;
  $\Delta_{dead} \leftarrow \Delta_{dead} \cup \delta_{dead}$;
end
$\delta_{loop} \leftarrow CreateNewTransition(q_{dead}, \top \downarrow \emptyset \rightarrow q_{dead})$;
$\Delta_{dead} \leftarrow \Delta_{dead} \cup \delta_{loop}$;
$Q_{comp} \leftarrow A.Q \cup \{q_{dead}\}$;
$q_{comp,s} \leftarrow A.q_s$;
$Q_{comp,f} \leftarrow A.Q \setminus A.Q_f$;
$R_{comp} \leftarrow A.R$;
$\Delta_{comp} \leftarrow A.\Delta \cup \Delta_{dead}$;
$A_{complement} \leftarrow (Q_{comp}, q_{comp,s}, Q_{comp,f}, R_{comp}, \Delta_{comp})$;
`return` $A_{complement}$;

References

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

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

3 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 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 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 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 Sase open source system. https://github.com/haopeng/sase. [Online; accessed 23-May-2024].

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

9 Elias Alevizos, Alexander Artikis, and George Paliouras. Wayeb: a tool for complex event forecasting. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 26–35. EasyChair, 2018.

10 Elias Alevizos, Alexander Artikis, and Georgios Paliouras. Symbolic automata with memory: a computational model for complex event processing. *CoRR*, abs/1804.09999, 2018.

11 Elias Alevizos, Alexander Artikis, and Georgios Paliouras. Symbolic register automata for complex event recognition and forecasting. *CoRR*, abs/2110.04032, 2021.

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

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

14 Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a complex event recognition engine. *CoRR*, abs/2111.04635, 2021.

15 Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a complex event recognition engine. *Proc. VLDB Endow.*, 15(9):1951–1964, 2022.

16 Badrish Chandramouli, Jonathan Goldstein, and David Maier. High-performance dynamic pattern matching over disordered streams. *Proc. VLDB Endow.*, 3(1):220–231, 2010.

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

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

19 Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. Symbolic register automata. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019.

20 Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 47–67. Springer, 2017.

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

22 Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 627–644. Springer, 2006.

23 Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422. www.cidrdb.org, 2007.

24 Christophe Dousson and Pierre Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *IJCAI*, pages 324–329, 2007.

25 Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010.

26 Malik Ghallab. On chronicles: Representation, on-line recognition and learning. In *KR*, pages 597–606. Morgan Kaufmann, 1996.

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

28 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *ICDT*, volume 127 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

29 Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. On the expressiveness of languages for complex event recognition. In *ICDT*, volume 155 of *LIPIcs*, pages 15:1–15:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

30 Sylvain Hallé. From complex event processing to simple event processing. *CoRR*, abs/1702.08051, 2017. URL: `http://arxiv.org/abs/1702.08051`.

31 Shawn Hedman. *A First Course in Logic: An introduction to model theory, proof theory, computability, and complexity*. Oxford University Press Oxford, 2004.

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

33 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

34 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

35 Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. Index-accelerated pattern matching in event stores. In *SIGMOD Conference*, pages 1023–1036. ACM, 2021.

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

37 Leonid Libkin and Domagoj Vrgoc. Regular expressions for data words. In *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2012.

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

39 Periklis Mantenoglou, Dimitrios Kelesis, and Alexander Artikis. Complex event recognition with allen relations. In *KR*, pages 502–511, 2023.

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

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

42 Dusan Petkovic. Specification of row pattern recognition in the SQL standard and its implementations. *Datenbank-Spektrum*, 22(2):163–174, 2022.

43 Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.

44 Efthimis Tsilionis, Alexander Artikis, and Georgios Paliouras. Incremental event calculus for run-time reasoning. *J. Artif. Intell. Res.*, 73:967–1023, 2022.

45 Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

46 Margus Veanes. Applications of symbolic finite automata. In *CIAA*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23. Springer, 2013.

47 Margus Veanes, Nikolaj Bjørner, and Leonardo Mendonça de Moura. Symbolic automata constraint solving. In *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 640–654. Springer, 2010.

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

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

50 Erkang Zhu, Silu Huang, and Surajit Chaudhuri. High-performance row pattern recognition using joins. *Proc. VLDB Endow.*, 16(5):1181–1194, 2023.