Symbolic Register Automata for Complex Event Recognition and Forecasting

Elias Alevizos
Department of Informatics,
National and Kapodistrian University of Athens, Greece
Institute of Informatics & Telecommunications,
National Center for Scientific Research “Demokritos”, Greece
ilalev@di.uoa.gr,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 propose an automaton model which is a combination of symbolic and register automata, i.e., we enrich symbolic automata with memory. We call such automata Symbolic Register Automata (SRA). SRA extend the expressive power of symbolic automata, by allowing Boolean formulas to be applied not only to the last element read from the input string, but to multiple elements, stored in their registers. SRA also extend register automata, by allowing arbitrary Boolean formulas, besides equality predicates. We study the closure properties of SRA under union, intersection, concatenation, Kleene closure, complement and determinization and show that SRA, contrary to symbolic automata, 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 SRA can be used in Complex Event Recognition 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. We also show how the behavior of SRA, as they consume streams
of events, can be given a probabilistic description with the help of prediction suffix trees. This allows us to go one step beyond Complex Event Recognition to Complex Event Forecasting, where, besides detecting complex patterns, we can also efficiently forecast their occurrence.

**Keywords:** Finite Automata, Regular Expressions, Complex Event Processing, Symbolic Automata, Register Automata, Variable-order Markov Models

### 1. Introduction

A Complex Event Recognition (CER) system takes as input a stream of 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 [1, 2, 3]. Typically, an event has the structure of a tuple of values which might be numerical or categorical. Since time is of critical importance for CEP, 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. Alternatively, these constraints might 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. The input to a CER system thus consists of two main components: a stream of events, also called simple derived events (SDEs); and a set of patterns that define relations among the SDEs. Instances of pattern satisfaction are called Complex Events (CEs). The output of the system is another stream, composed of the detected CEs. CEs must often be detected with very low latency, which, in certain cases, may even be in the order of a few milliseconds [2, 4, 5].

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, 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 [6, 7, 3]. They resemble both register automata [8], through their ability to store events, and symbolic automata [9], 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 automata is that their properties have not been systematically investigated, as is the case with models derived directly from the field of languages and automata (see [10] 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 [10]. Its advantage is that it provides a logic for CER patterns, with simple 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 [9]. 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.

We propose 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 Automata (SRA). SRA extend the expressive power of symbolic and register automata, by allowing for more complex patterns to be defined and detected on a stream of events. We also present a language with which we can define patterns for complex events that can then be translated to SRA. We call such patterns Symbolic Regular Expressions with Memory (SREM), as an extension of the work presented in [11], where Regular Expression with Memory (REM) are defined and investigated. REM are extensions of classical regular expressions with which we allow some of the terminal symbols of an expression to be stored and later be compared for (in)equality. SREM allow for more complex conditions to be used, besides those of (in)equality.

We then show how SREM and SRA may be used in order to perform Complex Event Forecasting (CEF). Our solution allows a user to define a pattern for a complex event in the form of a SREM. It then constructs a probabilistic model for such a pattern in order to forecast, on the basis of an event stream, if and when a complex event is expected to occur. We use prediction suffix trees [12] [13] to learn a probabilistic model for the pattern and the SRA corresponding to this pattern. We have already presented how symbolic automata without registers may be combined with prediction suffix trees for the purpose of CEF [14]. We show here when and how symbolic automata with memory can be combined with prediction suffix trees for the same purpose. Prediction suffix trees fall under the class of the so-called variable-order Markov models. They are Markov models whose order (how deep into the past they can look for dependencies) can be increased beyond what is computationally possible with full-order models. They can do this by avoiding a full enumeration of every possible dependency and focusing only on “meaningful” dependencies. Efficient and early CEF would thus allow analysts to take proactive action when critical situations are expected to happen, e.g., to alert maritime authorities for the possible collision of vessels at sea.

The contributions of the paper may be summarized as follows:

- We present a language for CER, Symbolic Regular Expressions with Memory (SREM).
- We present a computational model for patterns written in SREM, Symbolic Register Automata (SRA), whose main feature is that it allows for relating 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.
Table 1: Example stream.

| type | T | T | T | H | H | T | ... |
|------|---|---|---|---|---|---|-----|
| id   | 1 | 1 | 2 | 1 | 1 | 2 | ... |
| value| 22| 24| 32| 70| 68| 33| ... |
| index| 1 | 2 | 3 | 4 | 5 | 6 | ... |

- We show that $SRA$ and $SREM$ are equivalent, i.e., they accept the same set of languages.
- We study the closure properties of $SRA$ (and $SREM$). We show that, in the general case, they are closed under the most usual 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 requiring deterministic automata, like the ones we will describe later for event forecasting, are not applicable.
- We show that, by using windows, $SRA$ 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 when attempting to find matches for a pattern.
- We show how $SRA$ with windows can be combined with Prediction Suffix Trees in order to perform CEF, thus extending our previous work from symbolic automata to symbolic register automata \[14\].

All proofs and complete algorithms may be found in the Appendix, Section \[10\]. Please, note that the results of this paper are presented with CER in mind. However, we need to stress that they are not restricted to CER. They are general results, applicable to any strings and not just to streams of events. In fact, one may treat CER as a special case of string processing. Thus, our contributions lie both in the more specific field of CER and in the more general one of formal languages and automata theory.

**Example 1.1.** We now introduce an example which will be used throughout the paper to provide intuition (borrowed from \[10\]). The example is that of a set of sensors taking temperature and humidity measurements, monitoring an area for the possible eruption of fires. A stream is a sequence of input events, where each such event is a tuple of the form $(type, id, value)$. The first attribute $(type)$ is the type of measurement: $H$ for humidity and $T$ for temperature. The second one $(id)$ is an integer identifier, unique for each sensor. It has a finite set of possible values. Finally, the third one $(value)$ is the real-valued measurement from a possibly infinite set of values. 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. ◦
The structure of the paper is as follows. In Section 2 we present the state-of-the-art in automata theory with respect to automaton models that can store elements and models that can use more complex conditions on their transitions than simple equality. In Section 3 we discuss extensively the grammar and the semantics of SREM. Next, in Section 4 we define SRA and the languages that they recognize. In Section 5 we show that SRA and SREM are equivalent. We show that SRA and SREM are closed under union, intersection, concatenation and Kleene-star, but not under complement and determinization. We also define windowed SREM and SRA and show that windows make SRA and SREM closed under complement and determinization. In Section 6 we discuss how SRA can be used for CER, whereas Section 7 briefly discusses some complexity issues. Subsequently, Section 8 discusses how SRA can be given a probabilistic description through the use of variable-order Markov models in order to perform CEF. We conclude with Section 9 where we summarize our contributions and discuss possible avenues for future work.

2. Related Work

Because of 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 [15, 16], Cayuga [6], TESLA [17] and SASE [7, 18]. There also exist systems that do not employ automata as their computational model, e.g., there are logic-based systems [19] or systems that use trees [20], 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. For an overview of CER languages, see [1], and for a general review of CER systems, see [3].

However, current CER systems do not have the full expressive power of regular expressions, e.g., SASE does not allow for nesting Kleene-star operators. Moreover, due to the various approaches implementing the basic operators and extensions in their own way, there is a lack of a common ground that could act as a basis for systematically understanding the properties of these automaton models. The abundance of different CER systems, employing various computational models and using various formalisms has recently led to some attempts at providing a unifying framework [10, 21]. Specifically, in [10], 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.

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 [8, 22, 23]. For a review, see [24]. 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 [8]. At each transition, a register automaton may choose to store its current input (more precisely, the current input’s data payload) to one of a finite set of registers. A transition is followed if the current input is equal with 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 [11, 25]. We build on the framework presented in [11, 25] 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 [26]. This initial idea has recently been expanded and more fully investigated in symbolic automata [27, 28, 9]. In this automaton model, 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. The work presented in [10, 29] may also be categorized under this class of “unary” symbolic automata (or transducers, to be more precise). 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 in this paper. We propose an automaton model, called *Symbolic Register Automata* (SRA), whose transitions can apply $n$-ary formulas/conditions (with $n > 1$) on multiple elements. SRA are thus 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.

We also show how this new automaton model can be given a probabilistic description in order to perform forecasting, i.e., predict the occurrence of a complex event before it is actually detected by the automaton. However, forecasting has not received much attention in the field of CER, despite the fact that it is an active research topic in various related research areas, such as time-series forecasting [30], sequence prediction [31, 12, 32, 33], temporal mining [34, 35, 36, 37] and event sequence prediction and point-of-interest recommendations through neural networks [38, 39]. These methods are powerful in predicting the next numerical value(s) in a time-series or the next input event(s) in a sequence of events, but they suffer from limitations that render them unsuitable for CEF. In CEF we are interested in both numerical and categorical values, related through complex patterns and involving multiple variables. Such patterns require a language to be defined, much like SQL in databases. Our goal is to forecast the occurrence of such complex events defined via patterns and not input events. Input event forecasting is actually not very useful for CER, since the majority of input events are ignored, without contributing to the detection of complex events. The number of complex events is typically orders of magnitude lower than that of input events.

Some conceptual proposals have acknowledged the need for CEF though [40, 41, 42]. In what follows, we briefly present the relatively few previous concrete attempts at CEF. The first such attempt at CEF was presented in [43], where a variant of regular expressions and automata was used to define complex event patterns, along with Markov chains. Each automaton state was mapped to a Markov chain state. Symbolic automata and Markov chains were again used in [44, 45]. The problem with these approaches is that they are essentially unable to encode higher-order dependencies, since high-order Markov chains may lead to a combinatorial explosion of the number of states. In [46], complex events were defined through transitions systems and Hidden Markov Models (HMM) were used to construct a probabilistic model. The observable variable of the HMM corresponded to the states of the
transition system. HMMs are in general more powerful than Markov chains, but, in practice, the may be hard to train ([31, 47]) and require elaborate domain modeling, since mapping a pattern to a HMM is not straightforward. In contrast, our approach constructs seamlessly a probabilistic model from a given CE pattern (declaratively defined). Knowledge graphs were used in in [43] to encode events and their timing relationships. Stochastic gradient descent was employed to learn the weights of the graph’s edges that determine how important an event is with respect to another target event. However, this approach falls in the category of input event forecasting, as it does not target complex events.

3. A Grammar for Symbolic Regular Expressions with Memory

Before presenting SRA, we first present a high-level formalism for defining CER patterns. We extend the work presented in [11], 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 (SREM).

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 SREM, 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. Finally, in Section 3.3 we provide a precise definition for SREM and their semantics.

3.1. Formulas and Models

In this section, we follow the notation and notions presented in [49]. 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 3.1. ($\mathcal{V}$-structure [49])**

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 3.2.** Using Example 1.1 we can define the following vocabulary

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

and the universe

$$\mathcal{U} = \{(T, 1, 22), (T, 1, 24), (T, 2, 32), (H, 1, 70), (H, 1, 68), (T, 2, 33)\}$$
We can also define an interpretation of $V$ by assigning each $c_i$ to an element of $U$, e.g., $c_1$ to $(T, 1, 22)$, $c_2$ to $(T, 1, 24)$, etc. $R$ may also be interpreted as $R(x, y) := x.id = y.id$, i.e., this binary relation contains all pairs of $U$ which have the same $id$. For example, $((T, 1, 22), (H, 1, 70)) \in R$ and $((T, 1, 22), (T, 2, 33)) \notin R$. If there are more (even infinite) tuples in a stream/string, then we would also need more constants (even infinite).

We extend the terminology from classical regular expressions to define characters, strings and languages. Elements of $U$ are called characters and finite sequences of characters are called strings. A set of strings $L$ constructed from elements of $U$ ($L \subseteq U^*$, where $^*$ denotes Kleene-star) is called a language over $U$. We can also define streams as follows. A stream $S$ is an infinite sequence $S = t_1, t_2, \cdots$, where each $t_i$ is a character ($t_i \in U$). 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 $V$-structure. We begin with the definition of terms.

Definition 3.3. (Term [49])
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 3.4. (Formula [49])
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 $\phi$ is a formula, $\neg \phi$ is also a formula.
- If $\phi_1$ and $\phi_2$ are formulas, $\phi_1 \land \phi_2$ is also a formula.
- If $\phi_1$ and $\phi_2$ are formulas, $\phi_1 \lor \phi_2$ is also a formula. ◀

Definition 3.5. ($V$-formula [49])
If $V$ is a vocabulary, then a formula in which every function, relation and constant is in $V$ is called a $V$-formula. ◀

Example 3.6. Continuing with our example, $R(c_1, c_4)$ is an atomic $V$-formula. $R(c_1, c_4) \land \neg R(c_1, c_3)$ is also a (complex) $V$-formula, where $V = \{ R, c_1, c_2, c_3, c_4, c_5, c_6 \}$. ◇

Notice that in typical definitions of terms and formulas (as found in [49]) 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 \( U \)) 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 \( V \)-structure.

**Definition 3.7. (Model of \( V \)-formulas [49])**

Let \( M \) be a \( V \)-structure and \( \phi \) a \( V \)-formula. We define \( M \models \phi \) (\( M \) models \( \phi \)) as follows:

- If \( \phi \) is atomic, i.e. \( \phi = P(t_1, \ldots, t_m) \), then \( M \models P(t_1, \ldots, t_m) \) iff the tuple \((a_1, \ldots, a_m)\) is in the subset of \( U^m \) assigned to \( P \), where \( a_i \) are the elements of \( U \) assigned to the terms \( t_i \).

- If \( \phi := \neg \psi \), then \( M \models \phi \) iff \( M \not \models \psi \).

- If \( \phi := \phi_1 \land \phi_2 \), then \( M \models \phi \) iff \( M \models \phi_1 \) and \( M \models \phi_2 \).

- If \( \phi := \phi_1 \lor \phi_2 \), then \( M \models \phi \) iff \( M \models \phi_1 \) or \( M \models \phi_2 \). ◀

**Example 3.8.** If \( M \) is the \( V \)-structure of our example, then \( M \models R(c_1, c_4) \), since \( c_1 \rightarrow (T, 1, 22), c_1 \rightarrow (H, 1, 70) \), and \(( (T, 1, 22), (H, 1, 70)) \in R \). We can also see that \( M \models R(c_1, c_4) \land \neg R(c_1, c_3) \), since \( c_3 \rightarrow (T, 2, 32) \) and \(( (T, 1, 22), (T, 2, 32)) \notin R \). ◇

### 3.2. Conditions

Based on the above definitions, we will now define conditions over registers. These will essentially be the \( n \)-ary guards on the transitions of SRA.

**Definition 3.9. (Condition)**

Let \( M \) be a \( V \)-structure always equipped with the unary relation \( \top \) for which it holds that \( u \in \top \), \( \forall u \in U \), i.e., this relation holds for all elements of the universe \( U \). Let \( R = \{r_1, \ldots, r_k\} \) 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 them register variables. A condition is essentially a \( V \)-formula, as defined above (Definition 3.4), where, instead of terms, we use register variables. A condition is defined by the following grammar:

- \( \top \) is a condition.
- \( P(r_1, \ldots, r_n) \), where \( r_i \in R \cup \{\sim\} \) and \( P \) an \( n \)-ary relation, is a condition.
- \( \neg \phi \) is a condition, if \( \phi \) is a condition.
- \( \phi_1 \land \phi_2 \) is a condition if \( \phi_1 \) and \( \phi_2 \) are conditions.
• $\phi_1 \lor \phi_2$ is a condition if $\phi_1$ and $\phi_2$ are conditions. ▶

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 3.10. (Valuation)**

A valuation on $R = \{r_1, \ldots, r_k\}$ is a partial function $v : R \rightarrow \mathcal{U}$. The set of all valuations on $R$ is denoted by $F(r_1, \ldots, r_k)$. $v[r_i \leftarrow u]$ denotes the valuation where we replace the content of $r_i$ with a new element $u$:

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

$v[W \leftarrow u]$, 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 $\phi$ if, for every register variable $r_i$ that appears in $\phi$, $v(r_i)$ is defined. ▶

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

We can now define the semantics of conditions, similarly to the way we defined models of $\mathcal{V}$-formulas in Definition 3.7. 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 3.11. (Semantics of conditions)**

Let $\mathcal{M}$ be a $\mathcal{V}$-structure, $u \in \mathcal{U}$ an element of the universe of $\mathcal{M}$ and $v \in F(r_1, \ldots, r_k)$ a valuation. We say that a condition $\phi$ 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 $(u, v) \not\models \psi$.

• $\phi := \phi_1 \land \phi_2$, $(u, v) \models \phi_1$ and $(u, v) \models \phi_2$.

• $\phi := \phi_1 \lor \phi_2$, $(u, v) \models \phi_1$ or $(u, v) \models \phi_2$. ▶

### 3.3. Symbolic Regular Expressions with Memory

We are now in a position to define Symbolic Regular Expressions with Memory $\text{SREM}$. We achieve this by combining conditions via the standard regular operators. Conditions act as terminal “symbols”, as the base case from which we construct more complex expressions.

**Definition 3.12. (Symbolic regular expression with memory ($\text{SREM}$))**

A symbolic regular expression with memory 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. $\epsilon$ and $\emptyset$ are $SREM$.

2. If $\phi$ is a condition (as in Definition 3.9), then $\phi$ is a $SREM$.

3. If $\phi$ is a condition, then $\phi \downarrow r_i$ is a $SREM$. 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 $SREM$, then $e_1 + e_2$ is also a $SREM$. This corresponds to disjunction.

5. If $e_1$ and $e_2$ are $SREM$, then $e_1 \cdot e_2$ is also a $SREM$. This corresponds to concatenation.

6. If $e$ is a $SREM$, then $e^*$ is also a $SREM$. This corresponds to Kleene-star.

In order to define the semantics of $SREM$, we need to define precisely how the contents of the registers may change. We thus need to define how a $SREM$, starting from a given valuation $v$ and reading a given string $S$, reaches another valuation $v'$.

**Definition 3.13. (Semantics of $SREM$)**

Let $e$ be a $SREM$ over a $\mathcal{V}$-structure $\mathcal{M}$ and a set of register variables $R = \{r_1, \ldots, r_k\}$, $S$ a string constructed from elements of the universe of $\mathcal{M}$ and $v, v' \in F(r_1, \ldots, r_k)$. We define the relation $(e, S, v) \vdash v'$ as follows (a textual explanation is provided after the formal definition):

1. $(\epsilon, S, v) \vdash v'$ iff $S = \epsilon$ and $v = v'$.

2. $(\phi, S, v) \vdash v'$ iff $\phi \neq \epsilon$, $S = u$, $(u, v) \models \phi$ and $v' = v$.

3. $(\phi \downarrow r_i, S, v) \vdash v'$ iff $S = u$, $(u, v) \models \phi$ and $v' = v[r_i \leftarrow u]$.

4. $(e_1 \cdot e_2, S, v) \vdash v'$ iff $S = S_1 \cdot S_2$, $(e_1, S_1, v) \vdash v''$ and $(e_2, S_2, v'') \vdash v'$.

5. $(e_1 + e_2, S, v) \vdash v'$ iff $(e_1, S, v) \vdash v'$ or $(e_2, S, v) \vdash v'$.

6. $(e^*, S, v) \vdash v'$ iff

   \[
   \begin{cases}
   S = \epsilon \text{ and } v' = v \\
   S = S_1 \cdot S_2 : (e, S_1, v) \vdash v'' \text{ and } (e^*, S_1, v'') \vdash v'
   \end{cases}
   \]

   In the first case, we have an $\epsilon$ $SREM$. It may reach another valuation only if it reads an $\epsilon$ 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 and the given register contents. Again, the registers do not change. 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. The fifth case is a disjunction. Finally, the sixth case implies that we must be able to break the initial string into multiple sub-strings such that each one of
these substring 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 SREM accepts (as in [11]). The language of a SREM contains all the strings with which we can reach a valuation, starting from the empty valuation, where all registers are empty.

Definition 3.14. (Language accepted by a SREM)

We say that \((e, S, v)\) infers \(v'\) if \((e, S, v) \vdash v'\). We say that \(e\) induces \(v\) on a string \(S\) if \((e, S, \#) \vdash v\), where \(\#\) denotes the valuation in which no \(v(r_i)\) is defined, i.e., all registers are empty. The language accepted by a SREM \(e\) is defined as \(L(e) = \{S \mid (e, S, \#) \vdash v\}\) for some valuation \(v\).

Example 3.15. As an example, consider the following SREM

\[
e_1 := (\text{TypeIsT} (\sim) \downarrow r_1) \cdot (\top)^* \cdot (\text{TypeIsH} (\sim) \land \text{EqualId} (\sim, r_1))
\]

where we assume that a) \(\text{TypeIsT}(x) := x.\text{type} = T\), b) \(\text{TypeIsH}(x) := x.\text{type} = H\) and c) \(\text{EqualId}(x,y) := x.\text{id} = y.\text{id}\). If we feed the string/stream of Table 1 to \(e_1\), then we will have the following. We will initially read the first element \((T, 1, 22)\). Since its type is \(T\), we will move on and store \((T, 1, 22)\) to register \(r_1\), i.e., we will move from the empty valuation where \(v(r_1) = \#\) to \(v'\), where \(v'(r_1) = (T, 1, 22)\). Then, the sub-expression \((\top)^*\) lets us skip any number of elements. We can thus skip the second and third elements without changing the register contents. Now, upon reading the fourth element \((H, 1, 70)\), there are two options. Either skip it again to read the fifth element or try to move on by checking the sub-expression \((\text{TypeIsH}(\sim) \land \text{EqualId}(\sim, r_1))\). This condition is actually satisfied, since the type of this element is indeed \(H\) and its \(id\) is equal to the \(id\) of the element store in \(r_1\). Thus, \(S_{1..4}\) is indeed accepted by \(e_1\). With a similar reasoning we can see that the same is also true for \(S_{1..5}\).

4. Symbolic Register Automata

In order to capture SREM, we propose Symbolic Register Automata (SRA), an automaton model equipped with memory and logical conditions on its transitions. The basic idea is the following. We add a set of registers \(R\) to an automaton in order to be able to store elements from the string/stream that will be used later in \(n\)-ary conditions. Each register can store at most one element. In order to evaluate whether to follow a transition or not, each transition is equipped with a guard, in the form of a condition. If the condition evaluates to true, then the transition is followed. Since a condition might be \(n\)-ary, with \(n>1\), the values passed to its arguments during evaluation may be either the current element or the contents of some registers, i.e., some past elements. In other words, the transition is also equipped with a register selection, i.e., a tuple of registers. Before evaluation, the automaton reads the contents of those registers, passes them as arguments to the condition and the condition is evaluated. Additionally, if, during a run of the automaton, a transition is followed, then the transition has the option to write the element that triggered it to some of the automaton’s registers. These are called its write registers, i.e., the registers whose contents may be changed by the transition. We also
allow for £-transitions, as in classical automata, i.e., transitions that are followed without consuming any elements and without altering the contents of the registers.

We now formally define SRA. To aid understanding, we present three separate definitions: one for the automaton itself, one for its configurations and one for its runs.

**Definition 4.1. (Symbolic Register Automaton)**
A symbolic register automaton (SRA) with \( k \) registers over a \( \mathcal{V} \)-structure \( \mathcal{M} \) is a tuple \( (Q, q_s, Q_f, R, \Delta) \) where

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

A transition \( \delta \in \Delta \) is a tuple \( (q, \phi, W, q') \), also written as \( q, \phi \downarrow W \to q' \), where

- \( q, q' \in Q \),
- \( \phi \) is a condition, as defined in Definition 3.9 or \( \phi = \epsilon \) and
- \( W \in 2^R \) are the write registers.

We will use the dot notation to refer to elements of tuples. For example, if \( A \) is a SRA, then \( A.Q \) is the set of its states. For a transition \( \delta \), we will also use the notation \( \delta.\text{source} \) and \( \delta.\text{target} \) to refer to its source and target states respectively.

**Example 4.2.** As an example, consider the SRA of Figure 1. Each transition is represented as \( \phi \downarrow W \), where \( \phi \) is its condition 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). The definitions for the conditions of the transitions are presented in a separate box, above the SRA. 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 event consumed from the stream ($\sim$) and passes it as argument to the unary formula $\phi_1$. If $\phi_1$ 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 event and the event stored in $r_1$ ($\sim$) and passes them to the binary formula $\phi_2$. The condition $\top$ (in the self-loop of $q_1$) is a unary condition that always evaluates to true and allows us to skip any number of events. The SRA of Figure [1] captures SREM [2]. °

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

**Definition 4.3. (Configuration of SRA)**
Assume a string $S = t_1, t_2, \cdots, t_l$ and a SRA $A$ consuming $S$. A configuration of $A$ 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 $A$ and

- $v$ the current valuation, i.e., the current contents of $A$’s registers.

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

- $\exists \delta : \delta.\text{source} = q, \delta.\text{target} = q', \delta.\phi = \epsilon, j' = j, v' = v$, i.e., if this is an $\epsilon$ transition, we move to the target state without changing the index or the registers’ contents.

- $\exists \delta : \delta.\text{source} = q, \delta.\text{target} = q', \delta.W = \emptyset, (t_j, v) \models \delta.\phi, j' = j + 1, v' = v$, 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.

- $\exists \delta : \delta.\text{source} = q, \delta.\text{target} = q', \delta.W \neq \emptyset, (t_j, v) \models \delta.\phi, j' = j + 1, v' = v[W \leftarrow t_j]$, 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 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 $\epsilon$-transition, we do not move the stream pointer and do not update the registers, but only move to the next state.

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

**Definition 4.4. (Run of SRA over string/stream)**
A run $\rho$ of a SRA $A$ 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 $q_{n+1} \in A.Q_f$. °
Example 4.5. A run of the SRA of Figure 1 while consuming the first four events from the stream of Table 1 is the following:

\[
[1, q_s, \#] \xrightarrow{\delta_{s,s}^{-1}} [2, q_1, (T, 1, 22)] \xrightarrow{\delta_1^{-1}} [3, q_1, (T, 1, 22)] \xrightarrow{\delta_1} [4, q_1, (T, 1, 22)] \xrightarrow{\delta_{1,f}} [5, q_f, (T, 1, 22)]
\]

Transition subscripts in this example refer to states of the SRA, 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. See also Figure 2. Run (3) is not the only run, since the SRA 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 SRA always remains in \( q_1 \) after its first transition. ⚫

Finally, we can define the language of a SRA as the set of strings for which the SRA has an accepting run, starting from an empty configuration.
Definition 4.6. (Language recognized by SRA)
We say that a SRA $A$ 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 $A$ over $S$, where $q_1 = A.q_s$ and $v_1 = \#$. The set of all strings accepted by $A$ is called the language recognized by $A$ and is denoted by $L(A)$.

5. Properties of Symbolic Register Automata

We now study the properties of SRA. First, we prove the equivalence of SRA and SREM. We then show that SRA and SREM are closed under union, intersection, concatenation and Kleene-start but not under complement and determinization. We can thus construct SREM and SRA 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 SRA cannot be used in cases where this might be required, as in CEF. We will discuss in Section 8 why determinization is important for CEF. If, however, we use an extra window operator, effectively limiting the length of strings accepted by a SRA, we can then show that closure under complement and determinization is also possible.

5.1. Equivalence of SREM and SRA

We first prove that, for every SREM there exists an equivalent SRA. The proof is constructive, similar to that for classical automata. For the inverse direction, i.e. converting a SRA to an equivalent SREM, we use the notion of generalized SRA. These are SRA which have complete SREM on their transitions. By incrementally removing states from the SRA, we are finally left with two states and the SREM which connects them is the SREM we are looking for.

We now show how, for each SREM we can construct an equivalent SRA. Equivalence between an expression $e$ and a SRA $A$ means that they recognize the same language, i.e., $L(e) = L(A)$. See Definitions 3.14 and 4.6.

Theorem 5.1. For every SREM $e$ there exists an equivalent SRA $A$, i.e., a SRA such that $L(e) = L(A)$.

Proof:
The complete SRA construction process and proof may be found in Appendix 10.1.

Example 5.2. Here, we present an example, to give the intuition. Let

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

be a SREM, where

$$\phi_1(x) := (x.type = T \wedge x.value < -40)$$

$$\phi_2(x) := (x.type = T \wedge x.value > 50)$$

$$\phi_3(x, y) := (x.type = T \wedge x.id = y.id)$$
With this expression, we want to monitor sensors for possible failures. We want to detect cases where a sensor records temperatures outside some range of values (first line of \textit{SREM} (4)) and continues to transmit measurements (second line), so that we are alerted to the fact that new measurements might not be trustworthy. The last condition is a binary formula, applied to both $\sim$ and $r_1$. Figure 3 shows the process for constructing the \textit{SRA} which is equivalent to \textit{SREM} (4).

The algorithm is compositional, starting from the base cases $e:=\phi$ or $e:=\phi \downarrow W$. 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 \textit{SREM} $e$, 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 \( \phi_1(x) := \left( x.\text{type} = T \land x.\text{value} < -40 \right) \) to \( q_{s,1} \rightarrow q_{f,1} \). To the third transition, we add the relevant binary condition \( \phi_3(x, y) := \left( x.\text{type} = T \land x.\text{id} = y.\text{id} \right) \). The + operator is handled by joining the SRA of the disjuncts through new states and \( \epsilon \)-transitions. See Figure 3b. The concatenation operator is handled by connecting the SRA of its sub-expressions through an \( \epsilon \)-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 \( \epsilon \)-transition.

We can also prove the inverse theorem, i.e., that every SRA can be converted to a SREM. To do so, however, we will need two lemmas. The first is the standard lemma about \( \epsilon \) elimination, stating that we can always eliminate all \( \epsilon \) transitions from a SRA to get an equivalent SRA with no \( \epsilon \) transitions.

**Lemma 5.3.** For every SRA \( A_{\epsilon} \) with \( \epsilon \) transitions there exists an equivalent SRA \( A_{\not\epsilon} \) without \( \epsilon \) transitions, i.e., a SRA such that \( \mathcal{L}(A_{\epsilon}) = \mathcal{L}(A_{\not\epsilon}) \).

**Proof:**
See Appendix 10.2.

The next lemma that we will require concerns the ability of SRA to write to multiple registers at the same time. The write registers of a transition \( \delta \) in Definition 4.1, \( \delta.W \), might not be a singleton. On the other hand, according to Definition 3.12, each terminal sub-expression in a SREM may write to at most one register. We can prove though that being able to write to multiple registers at the same time does not add any expressive power to SRA. Every SRA which can write to multiple registers can be converted to a SRA whose transitions can write to at most one register.

**Definition 5.4.** A SRA \( A \) is called a multi-register SRA if there exists a transition \( \delta \in A.\Delta \) such that \( |\delta.W| > 1 \), i.e., if there exists a transition that can write to multiple registers. A SRA \( A \) is called a single-register SRA if for all transitions \( \delta \in A.\Delta \) it holds that \( |\delta.W| \leq 1 \), i.e., if each transition can write to at most one register.

**Lemma 5.5.** For every multi-register SRA \( A_{mr} \) there exists an equivalent single-register SRA \( A_{sr} \), i.e., a single-register SRA such that \( \mathcal{L}(A_{mr}) = \mathcal{L}(A_{sr}) \).

**Proof:**
See Appendix 10.3.

We are now in a position to prove that every SRA can be converted to a SREM.
Theorem 5.6. For every SRA $A$ there exists an equivalent SREM $e$, i.e., a SREM such that $L(A) = L(e)$.

Proof: See Appendix 10.4.

\[ \square \]

5.2. Closure Properties of SREM/SRA

We now study the closure properties of SRA under union, intersection, concatenation, Kleene-star, complement and determinization. We first provide the definition for deterministic SRA. Informally, a SRA is said to be deterministic if, at any time, with the same input event, it can follow no more than one transition. The formal definition is as follows:

Definition 5.7. (Deterministic SRA ($d$SRA))

A SRA $A$ with $k$ registers $\{r_1, \cdots, r_k\}$ over a $\mathcal{V}$-structure $\mathcal{M}$ is deterministic if, for all transitions $q, \phi_1 \downarrow W_1 \rightarrow q_1 \in A.\Delta$ and $q, \phi_2 \downarrow W_2 \rightarrow q_2 \in A.\Delta$, if $q_1 \neq q_2$ then, for all $u \in \mathcal{M}.U$ and $v \in F(r_1, \cdots, r_k)$, $(u, v) \models \phi_1$ and $(u, v) \not\models \phi_2$ 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 SRA, at most one run may exist for every string/stream.

We now give the definition for closure under union, intersection, concatenation, Kleene-star, complement and determinization:

Definition 5.8. (Closure of SRA)

We say that SRA are closed under:

- union if, for every SRA $A_1$ and $A_2$, there exists a SRA $A$ such that $L(A) = L(A_1) \cup L(A_2)$, i.e., a string $S$ is accepted by $A$ iff it is accepted either by $A_1$ or by $A_2$.
- intersection if, for every SRA $A_1$ and $A_2$, there exists a SRA $A$ such that $L(A) = L(A_1) \cap L(A_2)$, i.e., a string $S$ is accepted by $A$ iff it is accepted by both $A_1$ and $A_2$.
- concatenation if, for every SRA $A_1$ and $A_2$, there exists a SRA $A$ such that $L(A) = L(A_1) \cdot L(A_2)$, i.e., $S$ is accepted by $A$ iff it can be broken into two sub-strings $S = S_1 \cdot S_2$ such that $S_1$ is accepted by $A_1$ and $S_2$ by $A_2$.
- Kleene-star if, for every SRA $A$, there exists a SRA $A^*$ such that $L(A^*) = (L(A))^*$, where $L^* = \bigcup_{i \geq 0} L^i$, i.e., $S$ is accepted by $A_*$ iff it can be broken into $S = S_1 \cdot S_2 \cdots$ such that each $S_i$ is accepted by $A$. 

• complement if, for every SRA \( A \), there exists a SRA \( A_c \) such that for every string \( S \) it holds that \( S \in \mathcal{L}(A) \Leftrightarrow S \notin \mathcal{L}(A_c) \).

• determinization if, for every SRA \( A \), there exists a dSRA \( A_d \) such that \( \mathcal{L}(A) = \mathcal{L}(A_d) \).

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

**Theorem 5.9.** SRA and SREM are closed under union, intersection, concatenation and Kleene-star.

**Proof:**
See Appendix [10.5] for complete proofs. For union, concatenation and Kleene-star the proof is essentially the proof for converting SREM to SRA (and we have already proven that SRA and SREM are equivalent). For intersection, we construct a new SRA \( A \) with \( Q = A_1.Q \times A_2.Q \). Then, for each \( q = (q_1, q_2) \in Q \) we add a transition \( \delta \) to \( q' = (q'_1, q'_2) \in Q \) if there exists a transition \( \delta_1 \) from \( q_1 \) to \( q'_1 \) in \( A_1 \) and a transition \( \delta_2 \) from \( q_2 \) to \( q'_2 \) in \( A_2 \). The write registers of \( \delta \) are \( W = \delta_1.W \cup \delta_2.W \), i.e., we use multi-register SRA. This new SRA can reach a final state only if both \( A_1 \) and \( A_2 \) reach their final states on a given string \( S \).

On the other hand, SRA are not closed under complement:

**Theorem 5.10.** SRA and SREM are not closed under complement.

**Proof:**
See Appendix [10.6]  

It is also not always possible to determinize them:

**Theorem 5.11.** SRA are not closed under determinization.

**Proof:**
See Appendix [10.7]  

SRA can thus be constructed from four basic operators (union, intersection, concatenation and Kleene-star) in a compositional manner, providing substantial flexibility and expressive power for CER applications. However, as is the case for register automata [8], SRA are not closed under complement, something which 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.

SRA are also not closed under determinization, a result which might seem discouraging. In the next section, we show that there exists a sub-class of SREM for which a translation to deterministic SRA is indeed possible. This is achieved if we apply a windowing operator and limit the length of strings accepted by SREM and SRA.
5.3. Windowed SREM/SRA

We can overcome the negative results about complement and determinization by using windows in SREM and SRA. 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 SREM.

Definition 5.12. (Windowed SREM)

Let \( e \) be a SREM over a \( \mathcal{V} \)-structure \( \mathcal{M} \) and a set of register variables \( R = \{ r_1, \ldots, r_k \} \), \( S \) a string constructed from elements of the universe of \( \mathcal{M} \) and \( v, v' \in F(r_1, \ldots, r_k) \). A windowed SREM (wSREM) is an expression of the form \( e' := e^{[1..w]} \), where \( w \in \mathbb{N}_1 \). We define the relation \( (e', S, v) \vdash v' \) as follows: \( (e, S, v) \vdash v' \) and \( |S| \leq w \).

The windowing operator does not add any expressive power to SREM. 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.

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

Lemma 5.13. For every windowed SREM there exists an equivalent unrolled SRA without any loops, i.e., a SRA where each state may be visited at most once.

Proof:
The full proof and the complete construction algorithm are presented in Appendix 10.9.

Algorithm 1: Constructing unrolled SRA for windowed SREM (simplified).

Input: Windowed SREM \( e' := e^{[1..w]} \)
Output: Deterministic SRA \( A_{e'} \) equivalent to \( e' \)
1. \( A_{e,e} \leftarrow \text{ConstructSRA}(e) \);
2. \( A_e \leftarrow \text{EliminateEpsilon}(A_{e,e}) \);
3. enumerate all walks of \( A_e \) of length up to \( w \); // Now unroll \( A_e \).
4. join walks through disjunction;
5. collapse common prefixes;

Example 5.14. Here, we provide only the general outline of the algorithm and an example. Consider, e.g., the following SREM:

\[
e_3 := (((\top)^* \cdot \text{TypeIsT}(\sim) \downarrow r_1) \cdot (\top)^* \cdot (\text{TypeIsH}(\sim) \land \text{EqualId}(\sim, r_1)))^{[1..w]} \tag{5}
\]

It can skip any number of events with the first sub-expression \((\top)^*\). Then it expects to find an event with type \( T \) and stores it to register \( r_1 \) (sub-expression \( \text{TypeIsT}(\sim) \downarrow r_1 \)). The third sub-expression is again \((\top)^*\), meaning that, after seeing a \( T \) event, we are allowed to skip events. Finally, with the
last sub-expression \((TypeIsH(\sim) \land EqualId(\sim, r_1))\), if some event after the \(T\) event is of type \(H\) and they have the same identifier, then the string is accepted, provided that its length is also at most \(w\). Figure 4b shows the steps taken for constructing the equivalent unrolled \(SRA\) for this expression. A simplified version of the unrolling algorithm is shown in Algorithm 1.

The construction algorithm first produces a \(SRA\) as usual, without taking the window operator into account (see line 1 of Algorithm 1). For our example, the result would be the \(SRA\) of Figure 4a (please, note that the automaton of Figure 4a is slightly different than that of Figure 1 due to the presence of the first \((\top)^*\) sub-expression). Then the algorithm eliminates any \(\epsilon\)-transitions (line 2). The next step is to use this \(SRA\) in order to create the equivalent unrolled \(SRA\) (\(uSRA\)). The rationale behind this step is that the window constraint essentially imposes an upper bound on the number of registers that would be required for a deterministic \(SRA\). For our example, if \(w=3\), then we know that we will need at least one register, if a \(T\) event is immediately followed by an \(H\) event. We will also need at most two registers, if two consecutive \(T\) events appear before an \(H\) event. The function of the \(uSRA\) is to create the number of registers that will be needed, through traversing the original \(SRA\). Algorithm 1 does this by enumerating all the walks of length up to \(w\) on the \(SRA\) graph, by unrolling any cycles. Lines 3–5 of Algorithm 1 show this process in a simplified manner. The \(uSRA\) for our example is shown in Figure 4b for \(w=3\). The actual algorithm does not perform an exhaustive enumeration, but incrementally creates the \(uSRA\), by using the initial \(SRA\) as a generator of walks. Every time we expand a walk, we add a new transition, a new state and possibly a new register, as clones of the original transition, state and register. In our example, we start by creating a clone of \(q_s\) in Figure 4a, also named \(q_s\) in Figure 4b. From the start state of the initial \(SRA\) we have two options. Either loop in \(q_s\) through the \(\top\) transition or move to \(q_1\) through the transition with the \(\phi_1\) condition. We can thus expand \(q_s\) of the \(uSRA\) with two new transitions: from \(q_s\) to \(q_t\) and from \(q_s\) to \(q_1\) in Figure 4b. We keep expanding the \(SRA\) this way until we reach final states and without exceeding \(w\). As a result, the final \(uSRA\) has the form of a tree, whose walks and runs are of length up to \(w\).

A \(uSRA\) then allows us to capture windowed expressions. Note though that the algorithm we presented above, due to the unrolling operation, can result in a combinatorial explosion of the number of states of the \(dSRA\), especially for large values of \(w\). Its purpose here was mainly to establish Lemma 5.13.

Having a \(uSRA\) makes it easy to subsequently construct a \(dSRA\):

**Theorem 5.15.** For every windowed \(SREM\) there exists an equivalent deterministic \(SRA\).

**Proof:**
The proof for determinization is presented in Appendix 10.10. It is constructive and the determinization algorithm is based on the powerset construction of the states of the non-deterministic \(SRA\). It is similar to the algorithm for symbolic automata [27, 9]. It does not add or remove any registers. It initially constructs the powerset of the states of the \(uSRA\). The members of this powerset will be the states of the \(dSRA\). It then tries to make each such new state, say \(q_d\), deterministic, by creating transitions with mutually exclusive conditions when they have the same output. The construction of these mutually exclusive conditions is done by gathering the conditions of all the transitions that have as their source a member of \(q_d\). Out of these conditions, the set of *minterms* is created, i.e., the mutually
E. Alevizos, A. Artikis, G. Paliouras / Symbolic Automata with Memory for CER/F

\[
\begin{align*}
\phi_1(x) &:= (x.\text{type} = T) \\
\phi_2(x, y) &:= (x.\text{type} = H \land x.\text{id} = y.\text{id})
\end{align*}
\]

(a) SRA for SREM \((\text{sREM})\) before unrolling.

(b) SRA for SREM \((\text{sREM})\) after unrolling cycles, for \(w = 3\) (whole SRA, black and light gray states) and \(w = 2\) (top 3 states in black).

(c) \(d\text{SRA}\) (only part of it), for \(w = 3\).

Figure 4: Constructing \(d\text{SRA}\) for SREM \((\text{sREM})\).
exclusive conjuncts constructed from the initial conditions, where each conjunct is a condition in its original or its negated form. A transition is then created for each minterm, with \( q_d \) being the source. Then, only one transition can be triggered, since these minterms are mutually exclusive.

**Example 5.16.** As an example, Figure 4c shows the result of converting the \( uSRA \) of Figure 4b to a \( dSRA \). We have simplified somewhat the conditions of each transition due to the presence of the \( T \) predicates in some of them. For example, the minterm \( \phi_1 \land \neg T \) for the start state is unsatisfiable and can be ignored while \( \phi_1 \land T \) may be simplified to \( \phi_1 \). The figure shows only part of the \( dSRA \) to avoid clutter. Note that some of the rightmost states may be further expanded. For example, state \( \{q_t,1,q_1,t\} \) (top right) can be expanded. With the minterm \( \phi_2(\sim,r_1) \land \neg \phi_2(\sim,r_2) \), it would go to the final state \( \{q_1,t,2\} \) (not shown in the figure).

Being able to derive a deterministic \( SRA \) is important for Complex Event Forecasting (CEF), since, as we will show, determinization is an important intermediate step in this task. A deterministic \( SRA \) essentially provides us with the “symbols” with which we populate a prediction suffix tree, the structure that captures the statistical properties of an input event stream.

We may now prove, as a corollary, that windowed \( SRA \) are also closed under complement:

**Corollary 5.17.** Windowed \( SRA \) are closed under complement.

**Proof:**
See Appendix [10.11]

6. **Streaming \( SRA \) for Complex Event Recognition**

We have thus far described how \( SREM \) and \( SRA \) can be applied to bounded strings that are known in their totality before recognition. A string is given to a \( SRA \) and an answer is expected about whether the whole string belongs to the automaton’s language or not. However, in CER/F we are required to handle continuously updated streams of events and detect instances of \( SREM \) 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 \( SREM \) and \( SRA \) may work in a streaming setting (the discussion in this section develops along the lines presented in our previous work [14], with the difference that here we are concerned with symbolic automata with memory). 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 \( \Sigma \) 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
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 to report the indices $i$ at which a complex event is 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 an instance of a SREM $e$ is detected at $k$ iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in \mathcal{L}(e)$. In order to detect complex events of a SREM $e$ on a stream, we use a streaming version of SREM and SRA.

**Definition 6.1. (Streaming SREM and SRA)**

If $e$ is a SREM, then $e_s = \top^* \cdot e$ is called the streaming SREM (sSREM) corresponding to $e$. A SRA $A_{e_s}$ constructed from $e_s$ is called a streaming SRA (sSRA) 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 $m$, $1 \leq m \leq k$.

**Proposition 6.2.** If $S = t_1, t_2, \cdots$ is a stream of elements from a universe $\mathcal{U}$ of a $\mathcal{V}$-structure $\mathcal{M}$, where $t_i \in \mathcal{U}$ and $e$ is a SREM over $\mathcal{M}$, then, for every $S_{m..k}$, $S_{m..k} \in \mathcal{L}(e)$ iff $S_{1..k} \in \mathcal{L}(e_s)$ (and $S_{1..k} \in \mathcal{L}(A_{e_s})$).

**Proof:**

See Appendix [10.12] □

Note that sSREM and sSRA are just special cases of SREM and SRA respectively. Therefore, every result that holds for SREM and SRA also holds for sSREM and sSRA as well.

### 7. Notes on 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 SREM and SRA.

The complexity of these problems for SREM and SRA depends heavily on the nature of the conditions used as terminal expressions in SREM and as transition guards in SRA, e.g., the $\text{EqualId}(\sim, r_1)$ condition in SREM [5]. 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 SREM/SRA 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 [11]). Register automata are a special case of SRA, where the only allowed relations are the binary relations of equality and inequality. 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 SRA as well. On the other hand, the problem of non-emptiness is decidable but PSPACE-complete. The same problem for SRA is thus PSPACE-complete. Finally, the problem of membership is NP-complete. Therefore, it is also at least NP-complete for SRA. Note that membership is the most important problem for the purposes of CER/F, since in CER/F 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 SRA coincides with that for register automata.

If we focus our attention even further on windowed SRA, as is the case in CER/F, then we can estimate more precisely the complexity of processing a single event from a stream. This is the most important operation for CER/F. A windowed SRA can first be determinized (offline) to obtain a $dSRA$. Assume that the resulting $dSRA$ $A$ has $k$ 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 $k$ registers in order to evaluate the conditions. Therefore, the complexity of updating the state of the $dSRA$ $A$ is $O(c + k)$ (assuming that each register is accessed only once and its contents are provided to every condition which references that register).

8. Complex Event Forecasting with Markov Models

We now show how we can use the framework of SREM and SRA to perform Complex Event Forecasting (CEF). The main idea behind our forecasting method is the following: Given a pattern $e$ in the form of a SREM, we first construct an automaton. In order to perform event forecasting, we translate the SRA to an equivalent deterministic SRA. This $dSRA$ can then be used to learn a probabilistic model, typically a Markov model, that encodes dependencies among the events in an input stream. Note that deterministic SRA are important because they allow us, as we will show, to produce a stream of “symbols” from the initial stream of events. By using deterministic SRA, we can map each input event to a single symbol and then use this derived stream of symbols to learn a Markov model. With non-deterministic SRA each element/event from the string/stream may trigger multiple transitions and thus such a mapping is not possible. The probabilistic model is learned from a portion of the input stream which acts as a training dataset and it is then used to derive forecasts about the expected occurrence of the complex event encoded by the automaton. After learning a model, we need to estimate the so-called waiting-time distributions for each state of our automaton. These distributions let us know the probability of reaching a final state from any other automaton state in $k$ events from now. These distributions are then used to estimate forecasts, which generally have the form of an interval within which a complex event has a high probability of occurring.

We discern three cases and present them in order of increasing complexity:
• We have only unary conditions applied to the last event and an arbitrary (finite or infinite) universe. In this case, we do not need registers.

• We have \( n \) -ary conditions (with \( n \geq 1 \)) and a finite universe. In this case, registers are helpful, but may not be necessary. If we have a register automaton \( A \) and a finite universe \( \mathcal{U} \), we can always create an automaton \( A_U \) with states \( A.Q \times \mathcal{U} \) and appropriate transitions so that \( A_U \) is equivalent to \( A \) but has no registers. Its states can implicitly remember past elements.

• The most complex case is when we have \( n \) -ary conditions and an infinite universe, as is typically assumed in CER/F. Registers are necessary in this case.

8.1. SREM with unary conditions

As a first step, we assume that the given SREM contains only unary conditions. The universe in this case may be either finite or infinite. We have already presented how this case can be handled in our previous work \([14]\). We will present here only a high-level overview of our method and then discuss how it can be adjusted in order to accommodate the other two cases.

Before discussing how a dSRA can be described by a Markov model, we first discuss a useful result, which bears on the importance of being able to use deterministic automata. It can be shown that a dSRA always has an equivalent deterministic classical automaton, through a simple isomorphic mapping, retaining the exact same structure for the automaton and simply changing the conditions on the transitions with symbols \([50]\). This result is important for two reasons: a) it allows us to use methods developed for classical automata without having to always prove that they are indeed applicable to symbolic automata as well, and b) it will help us in simplifying our notation, since we can use the standard notation of symbols instead of predicates. This result implies that, for every run \( \varrho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k + 1, q_{k+1}, v_{k+1}] \) followed by a dSRA \( A_s \) by consuming a symbolic string (or stream of events) \( S \), the run that the equivalent classical automaton \( A_c \) follows by consuming the induced string \( S' \) is also \( \varrho' = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k + 1, q_{k+1}, v_{k+1}] \), i.e., \( A_c \) follows the same copied/renamed states and the same copied/relabeled transitions. We can then use symbols and strings (lowercase letters to denote symbols), as in classical theories of automata, bearing in mind that, in our case, each symbol always corresponds to a condition. Details may be found in \([14]\).

This equivalent deterministic classical automaton can be used to convert a string/stream of elements/events to a string/stream of symbols. Since each element may trigger only a single transition, the initial string of elements may be mapped to a string of symbols. Each transition \( \delta_i \) from the initial run \( \varrho \) corresponds to a transition \( \delta_i \) from run \( \varrho' \). Since each transition from \( \varrho' \) corresponds to a single symbol, we can map the whole stream of input events to a single string of symbols. This means that we can use techniques developed in the context of deterministic classical automata and apply them to our case. One such class of techniques concerns the question of how we can build a probabilistic model that captures the statistical properties of the streams to be processed by an automaton. Such a model would allow us to make inferences about the automaton’s expected behavior as it reads event streams.
We have proposed the use of a variable-order Markov model (VMM) \cite{31,12,13,32,33}. Compared to fixed-order Markov models, VMMs allow us to increase their order \( m \) (how many events they can remember) to higher values and thus capture longer-term dependencies, which can lead to a better accuracy.

The idea behind VMMs is the following: let \( \Sigma \) denote an alphabet, \( \sigma \in \Sigma \) a symbol from that alphabet and \( s \in \Sigma^m \) a string of length \( m \) of symbols from that alphabet. The aim is to derive a predictor \( \hat{P} \) from the training data such that the average log-loss on a test sequence \( S_{1..k} \) is minimized. The loss is given by 
\[
    l(\hat{P}, S_{1..k}) = -\frac{1}{T} \sum_{i=1}^{k} \log \hat{P}(t_i | t_1 \cdots t_{i-1}).
\]

Minimizing the log-loss is equivalent to maximizing the likelihood 
\[
    \hat{P}(S_{1..k}) = \prod_{i=1}^{k} \hat{P}(t_i | t_1 \cdots t_{i-1}).
\]

The average log-loss may also be viewed as a measure of the average compression rate achieved on the test sequence \cite{31}. The mean (or expected) log-loss 
\[
    -E_P(\log \hat{P}(S_{1..k}))
\]

is minimized if the derived predictor \( \hat{P} \) is indeed the actual distribution \( P \) of the source emitting sequences.

For fixed-order Markov models, the predictor \( \hat{P} \) is derived through the estimation of conditional distributions \( \hat{P}(\sigma | s) \), with \( m \) constant and equal to the assumed order of the Markov model (\( \sigma \) is a single symbol and \( s \) a string of length \( m \)). VMMs, on the other hand, relax the assumption of \( m \) being fixed. The length of the “context” \( s \) may vary, up to a maximum order \( m \), according to the statistics of the training dataset. By looking deeper into the past only when it is statistically meaningful, VMMs can capture both short- and long-term dependencies.

We use Prediction Suffix Trees (PST), as described in \cite{12,13}, as our VMM of choice. Assuming that we have derived an initial predictor \( \hat{P} \) (by scanning the training dataset and estimating various empirical conditional probabilities), the learning algorithm in \cite{12} starts with a tree having only a single node, corresponding to the empty string \( \epsilon \). Then, it decides whether to add a new context/node \( s \) by checking whether it is “meaningful enough” to expand to \( s \). This is achieved by checking whether there exists a significant difference between the conditional probability of a symbol \( \sigma \) given \( s \) and the same probability given the shorter context suffix \( s \) (suffix \( s \) is the longest suffix of \( s \) different than \( s \)). A detailed description of how we use PST to perform forecasting may be found in \cite{14}.

Our goal is to use the PST in order to to calculate the so-called waiting-time distribution for every state \( q \) of the automaton \( A \). The waiting-time distribution is the distribution of the index \( n \), given by the waiting-time variable \( W_q = \inf \{ n : Y_0, Y_1, ..., Y_n \} \), where \( Y_0 = q, Y_i \in A.Q \setminus A.Q_f \) for \( i \neq n \) and \( Y_n \in A.Q_f \). Thus, waiting-time distributions give us the probability to reach a final state from a given state \( q \) in \( n \) transitions from now.

**Example 8.1.** We provide here the intuition through an example. Figure 5a shows an example of a deterministic automaton. Note that we use symbols on the transitions, which, as explained, essentially correspond to conditions. Figure 5b shows a PST which could be constructed from the automaton of Figure 5a and a given training dataset, with \( m = 3 \). This PST is read as follows. Consider its left-most node, \( aa, (0.75, 0.25) \). This means that the probability of encountering an \( a \) symbol, given that the last two symbols are \( aa \), is 0.75. The probability of seeing \( b \), on the other hand, is 0.25. The order of this node is 2. It has not been further expanded to yet another deeper level, because it was estimated that such an expansion would be statistically insignificant. For example, the probability \( P(a \mid baa) \) might still be very close to 0.75 (e.g., 0.747). If the same is true for \( P(a \mid aaa) \), then this means that the probability of seeing \( a \) is not significantly affected by expanding to contexts of length
3. If a similar statistical insignificance can be established for the probability of $b$, then it does not make sense to expand the node, since its children would not provide us with more information.

Figure 5c illustrates how we can estimate the probability for any future sequence of states of the $dSRA$ $A$ of Figure 5a, using the distributions of the $PST$ $T$ of Figure 5b and thus how we can calculate the waiting-time distributions. We first assume that, as the system processes events from the input stream, besides feeding them to $A$, it also stores them in a buffer that holds the $m$ most recent events, where $m$ is equal to the maximum order of the $PST$ $T$. After updating the buffer with a new event, the system traverses $T$ according to the contents of the buffer and arrives at a leaf $l$ of $T$. Let us now assume that, after consuming the last event, $A$ is in state 1 in Figure 5a and $T$ has reached its left-most node, $aa$, $(0.75, 0.25)$ in Figure 5b. This is shown as the left-most node also in Figure 5c. Each node in this figure has two elements: the first one is the state of $A$ and the second the node of $T$, starting with $\{1, aa\}$ as our current “configuration”. Each node has two outgoing edges, one for $a$ and one for $b$, indicating what might happen next and with what probability. For example, from the left-most node of Figure 5c, we know that, according to $T$, we might see $a$ with probability $0.75$ and $b$ with probability $0.25$. If we do encounter $b$, then $A$ will move to state 2 and $T$ will reach leaf $b$, $(0.5, 0.5)$. This is shown in Figure 5c as the white node $\{2, b\}$. This node has a double border to indicate that $A$ has reached a final state.

In a similar manner, we can keep expanding this tree into the future and use it to estimate the waiting-time distribution for its node $\{1, aa\}$, i.e., the distribution for state 1 of the automaton of Figure 5a when we know that the last two read symbols are $aa$. In order to estimate the probability of reaching a final state for the first time in $k$ transitions, we first find all the paths of length $k$ which start from the original node and end in a final state without including another final state. In our example of Figure 5c if $k = 1$, then the path from $\{1, aa\}$ to $\{2, b\}$ is such a path and its probability is 0.25. Thus, $P(W_{\{1,aa\}} = 1) = 0.25$. For $k = 2$, the path with the purple nodes leads to a final state after 2 transitions. Its probability is $0.75 * 0.25 = 0.1875$, i.e., the product of the probabilities of the path edges. Thus, $P(W_{\{1,aa\}} = 2) = 0.1875$. If there were more such alternative paths, we would have to add their probabilities.

We can use the waiting-time distributions to produce various kinds of forecasts. In the simplest case, we can perform regression forecasting where we select the future point with the highest probability and return this point as a forecast. Alternatively, we may also perform classification forecasting, if our goal is to determine how likely it is that a CE will occur within the next $w$ input events. We can sum the probabilities of the first $w$ points of a distribution and if this sum exceeds a given threshold we emit a “positive” forecast, meaning that a CE is indeed expected to occur; otherwise a “negative” forecast is emitted, meaning that no CE is expected.

8.2. $SREM$ with n-ary conditions on a finite universe

Thus far, we have described how we can perform CEF when we only have unary conditions and a finite or infinite universe. In this case, we create the deterministic automaton and use it to generate a stream of symbols with which we can learn a $PST$. Note that, when we only have unary conditions and thus no need for registers, $SRA$ are in essence equivalent to symbolic automata. Symbolic automata are determinizable and closed under complement, without requiring a window. Thus, the method
E. Alevizos, A. Artikis, G. Paliouras / Symbolic Automata with Memory for CER/F

(a) Example automaton $A$. (b) Example $PST$ $T$ for the automaton $A$.

(c) Future paths followed by automaton $A$ and $PST$ $T$ starting from state 1 of $A$ and node $aa$ of $T$. Purple nodes correspond to the only path of length $k = 2$ that leads to a final state. Pink nodes are pruned. Nodes with double borders correspond to final states of $A$.

Figure 5: Example of estimating waiting-time distributions.
described above for forecasting applies to every SRA with unary conditions, regardless of whether a windowing operator is present.

The next case is when we have a finite universe and \( n \)-ary conditions, where \( n \geq 1 \). We can follow the same process as described above, with one important difference. Since the universe \( \mathcal{U} \) is finite, we can directly map each element of \( \mathcal{U} \) to a symbol. Therefore, the PST \( T \) can be constructed directly from the elements of \( \mathcal{U} \). In practice, however, if the cardinality of \( \mathcal{U} \) is high and we have a windowed SREM, it might be preferable to use the conditions of the dSRA \( A \), if their number is significantly lower. A PST with too many symbols can quickly become hard to manage as we increase its order \( m \) and it is thus advisable to avoid increasing recklessly the size of its alphabet.

**Example 8.2.** For example, assume that the values for humidity and temperature take only discrete values (low, high) and that we only have two sensors. Then

\[
\mathcal{U} = \{(T, 1, \text{low}), (T, 1, \text{high}), (T, 2, \text{low}), (T, 2, \text{high}), (H, 1, \text{low}), (H, 1, \text{high}), (H, 2, \text{low}), (H, 2, \text{high})\}
\]

We can then map \((T, 1, \text{low})\) to \(a\), \((T, 1, \text{high})\) to \(b\), etc. However, if we have an automaton that only checks whether a measurement comes from the same sensor as a measurement stored in a register, like \( \text{EqualId}(\sim, r_1) \), then we do not need 8 symbols. We can use only \(a\) and \(b\), with \(a\) corresponding to \( \text{EqualId}(\sim, r_1) \) and \(b\) to \( \neg\text{EqualId}(\sim, r_1) \). The automaton is able to convert a stream/string \( S \) constructed from \( \mathcal{U} \) to a stream/string of \(a\) and \(b\) symbols, which can then be used to construct a PST.

\[\diamondsuit\]

### 8.3. SREM with \( n \)-ary conditions on a infinite universe

Finally, the most general case is when we have an infinite universe and \( n \)-ary conditions. In this case, the applicability of our method is necessarily restricted to windowed SREM and SRA. We can construct a deterministic SRA (which, by definition, has only a single run) from the windowed SREM and use this dSRA to generate a (single) sequence of symbols from a training dataset. This sequence can then be used to learn a PST. Then, the dSRA and the PST can be combined, as already described above, to estimate the waiting-time distributions and the forecasts.

**Example 8.3.** As an example, Figure 6 shows the deterministic classical automaton that can be constructed for SREM (2) and the dSRA of Figure 4c (note that Figure 4c and thus Figure 6 do not depict automata in their totality, but only part of them). Assume that we use the stream of Table 1 as a training dataset. We feed it to the automaton of Figure 6. Upon reading the first input event, \((T, 1, 22)\), transition \(a\) is triggered. Since the automaton is deterministic, this is the only triggered transition. Thus, \((T, 1, 22)\) is mapped to \(a\) and the automaton moves to state \(\{q_t, q_1\}\). With the second event, \((T, 1, 24)\), transition \(f\) is triggered. Thus, \((T, 1, 24)\) is mapped to \(f\). We repeat this process until the whole stream of events has been mapped to a stream of symbols, \(S = a, f, \cdots\). \(S\) may now be used to learn a PST of a given maximum order. This PST, along with the automaton of Figure 6, can be used to estimate the waiting-time distributions, as in the example of Figure 5c. \[\diamondsuit\]
9. Summary & Future Work

We presented an automaton model, \( SRA \), that can act as a computational model for patterns with \( n \)-ary conditions (\( n \geq 1 \)), which are quintessential for practical CER/F applications. \( SRA \) thus extend the expressive power of symbolic automata. They also extend the expressive power of register automata, through the use of conditions that are more complex than (in)equality predicates. \( SRA \) 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. This is not the case though for complement/negation. We showed that complement may be used and determinization is also 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 \( SRA \), the cost of updating the state of such an automaton after reading a single element is linear in the number of registers and conditions. We then described how prediction suffix trees may be used to provide a probabilistic description for the behavior of \( SRA \). Prediction suffix trees can look deep into the past and make accurate inferences about the future behavior of \( SRA \), thus allowing us to forecast when a complex event is expected to occur.

As a next step, we intend to implement the proposed framework for CER/F, extending our open-source engine, Wayeb\(^1\), which currently supports only unary conditions. We also intend to investigate in the future the possibility of providing more precise complexity results for \( SREM \) and \( SRA \), both from the point of view of formal languages and from the point of view of CER/F, where some extra constraints may exist. For example, besides updating the state of an automaton after reading a new element, we may also need to take into account the time required to report the input events contributing to the detection of a complex event (for more details about this kind of complexity, please consult

\(^1\)Wayeb source code: [https://github.com/ElAlev/Wayeb](https://github.com/ElAlev/Wayeb)
where a model for evaluating efficiency in CER is presented, along with complexity results for symbolic transducers without memory). In the future, we intend to investigate the complexity of decision problems for SRA from this point of view.

Acknowledgements

This work has received funding from the EU Horizon 2020 program INFORE under grant agreement No 825070 and from the European Horizon 2020 program ARIADNE under grant agreement No 871464.

References

[1] Giatrakos N, Alevizos E, Artikis A, Deligiannakis A, Garofalakis MN. Complex event recognition in the Big Data era: a survey. VLDB J., 2020. 29(1):313–352. doi:10.1007/s00778-019-00557-w. URL https://doi.org/10.1007/s00778-019-00557-w

[2] Luckham DC. The power of events - an introduction to complex event processing in distributed enterprise systems. ACM, 2005. ISBN 978-0-201-72789-0.

[3] Cugola G, Margara A. Processing flows of information: From data stream to complex event processing. ACM Comput. Surv., 2012. 44(3):15:1–15:62. doi:10.1145/2187671.2187677. URL https://doi.org/10.1145/2187671.2187677

[4] Etzion O, Niblett P. Event Processing in Action. Manning Publications Company, 2010. ISBN 978-1-935182-21-4. URL http://www.manning.com/etzion/

[5] Hedtstück U. Complex event processing: Verarbeitung von Ereignismustern in Datenströmen. Springer Vieweg, Berlin, 2017.

[6] Demers AJ, Gehrke J, Panda B, Riedewald M, Sharma V, White WM. Cayuga: A General Purpose Event Monitoring System. In: Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings. www.cidrdb.org, 2007 pp. 412–422. URL http://cidrdb.org/cidr2007/papers/cidr07p47.pdf

[7] Agrawal J, Diao Y, Gyllstrom D, Immerman N. Efficient pattern matching over event streams. In: Wang JT (ed.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008. ACM, 2008 pp. 147–160. doi:10.1145/1376616.1376634. URL https://doi.org/10.1145/1376616.1376634

[8] Kaminski M, Francez N. Finite-Memory Automata. Theor. Comput. Sci., 1994. 134(2):329–363. doi:10.1016/0304-3975(94)90242-9. URL https://doi.org/10.1016/0304-3975(94)90242-9

[9] D’Antoni L, Veanes M. The Power of Symbolic Automata and Transducers. In: Majumdar R, Kunčak V (eds.), Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, volume 10426 of Lecture Notes in Computer Science. Springer, 2017 pp. 47–67. doi:10.1007/978-3-319-63387-9_3. URL https://doi.org/10.1007/978-3-319-63387-9_3
[10] Grez A, Riveros C, Ugarte M. A Formal Framework for Complex Event Processing. In: Barceló P, Calautti M (eds.), 22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal, volume 127 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019 pp. 5:1–5:18. doi:10.4230/LIPIcs.ICDT.2019.5. URL https://doi.org/10.4230/LIPIcs.ICDT.2019.5

[11] Libkin L, Tan T, Vrgoc D. Regular expressions for data words. J. Comput. Syst. Sci., 2015. 81(7):1278–1297. doi:10.1016/j.jcss.2015.03.005. URL https://doi.org/10.1016/j.jcss.2015.03.005

[12] Ron D, Singer Y, Tishby N. The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length. Mach. Learn., 1996. 25(2-3):117–149. doi:10.1023/A:1026490906255. URL https://doi.org/10.1023/A:1026490906255

[13] Ron D, Singer Y, Tishby N. The Power of Amnesia. In: Cowan JD, Tesauro G, Alspector J (eds.), Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]. Morgan Kaufmann, 1993 pp. 176–183. URL http://papers.nips.cc/paper/723-the-power-of-amnesia

[14] Alevizos E, Artikis A, Paliouras G. Complex event forecasting with prediction suffix trees. VLDB J., 2021. doi:https://doi.org/10.1007/s00778-021-00698-x. URL https://doi.org/10.1007/s00778-021-00698-x

[15] Ghallab M. On Chronicles: Representation, On-line Recognition and Learning. In: Aiello LC, Doyle J, Shapiro SC (eds.), Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR’96), Cambridge, Massachusetts, USA, November 5-8, 1996. Morgan Kaufmann, 1996 pp. 597–606.

[16] Dousson C, Maigat PL. Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization. In: Veloso MM (ed.), IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. 2007 pp. 324–329. URL http://ijcai.org/Proceedings/07/Papers/050.pdf

[17] Cugola G, Margara A. TESLA: a formally defined event specification language. In: Bacon J, Pietzuch PR, Sventek J, Çetintemel U (eds.), Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010. ACM, 2010 pp. 50–61. doi:10.1145/1827418.1827427. URL https://doi.org/10.1145/1827418.1827427

[18] Zhang H, Diao Y, Immerman N. On complexity and optimization of expensive queries in complex event processing. In: Dyreson CE, Li F, Özsu MT (eds.), International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM, 2014 pp. 217–228. doi:10.1145/2588555.2593671. URL https://doi.org/10.1145/2588555.2593671

[19] Artikis A, Sergot MJ, Paliouras G. An Event Calculus for Event Recognition. IEEE Trans. Knowl. Data Eng., 2015. 27(4):895–908. doi:10.1109/TKDE.2014.2356476. URL https://doi.org/10.1109/TKDE.2014.2356476

[20] Mei Y, Madden S. ZStream: a cost-based query processor for adaptively detecting composite events. In: Çetintemel U, Zdonik SB, Kossmann D, Tatbul N (eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. ACM, 2009 pp. 193–206. doi:10.1145/1559845.1559867. URL https://doi.org/10.1145/1559845.1559867

[21] Hallé S. From Complex Event Processing to Simple Event Processing. CoRR, 2017. abs/1702.08051. URL http://arxiv.org/abs/1702.08051
[22] Neven F, Schwentick T, Vianu V. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 2004. 5(3):403–435. doi:10.1145/1013560.1013562. URL https://doi.org/10.1145/1013560.1013562

[23] Bojanczyk M, David C, Muscholl A, Schwentick T, Segoufin L. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 2011. 12(4):27:1–27:26. doi:10.1145/1970398.1970403. URL https://doi.org/10.1145/1970398.1970403

[24] Segoufin L. Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik Z (ed.), Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings, volume 4207 of *Lecture Notes in Computer Science*. Springer, 2006 pp. 41–57. doi:10.1007/11874683_3. URL https://doi.org/10.1007/11874683_3

[25] Libkin L, Vrgoc D. Regular Expressions for Data Words. In: Bjørner N, Voronkov A (eds.), Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings, volume 7180 of *Lecture Notes in Computer Science*. Springer, 2012 pp. 274–288. doi:10.1007/978-3-642-28717-6_22. URL https://doi.org/10.1007/978-3-642-28717-6_22

[26] van Noord G, Gerdemann D. Finite State Transducers with Predicates and Identities. *Grammars*, 2001. 4(3):263–286. doi:10.1023/A:1012291501330. URL https://doi.org/10.1023/A:1012291501330

[27] Veanes M, Bjørner N, de Moura LM. Symbolic Automata Constraint Solving. In: Fermüller CG, Voronkov A (eds.), Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings, volume 6397 of *Lecture Notes in Computer Science*. Springer, 2010 pp. 640–654. doi:10.1007/978-3-642-16242-8_45. URL https://doi.org/10.1007/978-3-642-16242-8_45

[28] Veanes M. Applications of Symbolic Finite Automata. In: Konstantinidis S (ed.), Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings, volume 7982 of *Lecture Notes in Computer Science*. Springer, 2013 pp. 16–23. doi:10.1007/978-3-642-39274-0_3. URL https://doi.org/10.1007/978-3-642-39274-0_3

[29] Grez A, Riveros C, Ugarte M, Vansummeren S. On the Expressiveness of Languages for Complex Event Recognition. In: Lutz C, Jung JC (eds.), 23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark, volume 155 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020 pp. 15:1–15:17. doi:10.4230/LIPIcs.ICDT.2020.15. URL https://doi.org/10.4230/LIPIcs.ICDT.2020.15

[30] Montgomery DC, Jennings CL, Kulahci M. Introduction to time series analysis and forecasting. John Wiley & Sons, 2015.

[31] Begleiter R, El-Yaniv R, Yona G. On Prediction Using Variable Order Markov Models. *J. Artif. Intell. Res.*, 2004. 22:385–421. doi:10.1613/jair.1491. URL https://doi.org/10.1613/jair.1491

[32] Cleary JG, Witten IH. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. Commun.*, 1984. 32(4):396–402. doi:10.1109/TCOM.1984.1096090. URL https://doi.org/10.1109/TCOM.1984.1096090

[33] Willems FMJ, Shtarkov YM, Tjalkens TJ. The context-tree weighting method: basic properties. *IEEE Trans. Inf. Theory*, 1995. 41(3):653–664. doi:10.1109/18.382012. URL https://doi.org/10.1109/18.382012
[34] Vilalta R, Ma S. Predicting Rare Events In Temporal Domains. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan. IEEE Computer Society, 2002 pp. 474–481. doi:10.1109/ICDM.2002.1183991. URL https://doi.org/10.1109/ICDM.2002.1183991

[35] Laxman S, Tankasali V, White RW. Stream prediction using a generative model based on frequent episodes in event sequences. In: Li Y, Liu B, Sarawagi S (eds.), Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008. ACM, 2008 pp. 453–461. doi:10.1145/1401890.1401947. URL https://doi.org/10.1145/1401890.1401947

[36] Zhou C, Cule B, Goethals B. A pattern based predictor for event streams. Expert Syst. Appl., 2015. 42(23):9294–9306. doi:10.1016/j.eswa.2015.08.021. URL https://doi.org/10.1016/j.eswa.2015.08.021

[37] Cho C, Wu Y, Yen S, Zheng Y, Chen ALP. On-line rule matching for event prediction. VLDB J., 2011. 20(3):303–334. doi:10.1007/s00778-010-0197-3. URL https://doi.org/10.1007/s00778-010-0197-3

[38] Li Z, Ding X, Liu T. Constructing Narrative Event Evolutionary Graph for Script Event Prediction. In: Lang J (ed.), Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org, 2018 pp. 4201–4207. doi:10.24963/ijcai.2018/588. URL https://doi.org/10.24963/ijcai.2018/588

[39] Chang B, Park Y, Park D, Kim S, Kang J. Content-Aware Hierarchical Point-of-Interest Embedding Model for Successive POI Recommendation. In: Lang J (ed.), Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org, 2018 pp. 3301–3307. doi:10.24963/ijcai.2018/458. URL https://doi.org/10.24963/ijcai.2018/458

[40] Fülöp LJ, Beszédes Á, Toth G, Demeter H, Vidács L, Farkas L. Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics. In: Ivanovic M, Budimac Z, Radovanovic M (eds.), Balkan Conference in Informatics, 2012, BCI ‘12, Novi Sad, Serbia, September 16-20, 2012. ACM, 2012 pp. 26–31. doi:10.1145/2371316.2371323. URL https://doi.org/10.1145/2371316.2371323

[41] Engel Y, Etzion O. Towards proactive event-driven computing. In: Eyers DM, Etzion O, Gal A, Zdonik SB, Vincent P (eds.), Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011. ACM, 2011 pp. 125–136. doi:10.1145/2002259.2002279. URL https://doi.org/10.1145/2002259.2002279

[42] Christ M, Krumeich J, Kempa-Liehr AW. Integrating Predictive Analytics into Complex Event Processing by Using Conditional Density Estimations. In: Dijkman RM, Pires LF, Rinderle-Ma S (eds.), 20th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2016, Vienna, Austria, September 5-9, 2016. IEEE Computer Society, 2016 pp. 1–8. doi:10.1109/EDOCW.2016.7584363. URL https://doi.org/10.1109/EDOCW.2016.7584363

[43] Muthusamy V, Liu H, Jacobsen H. Predictive publish/subscribe matching. In: Bacon J, Pietzuch PR, Sventek J, Çetintemel U (eds.), Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010. ACM, 2010 pp. 14–25. doi:10.1145/1827418.1827423. URL https://doi.org/10.1145/1827418.1827423
Appendix

10.1. Proof of Theorem 5.1

Theorem 10.1. For every $SREM e$ there exists an equivalent $SRA A$, i.e., a $SRA$ such that $\mathcal{L}(e) = \mathcal{L}(A)$.

Proof:
For a SREM \( e \) and valuations \( v, v' \), let \( \mathcal{L}(e, v, v') \) denote all strings \( S \) such that \( (e, S, v) \vdash v' \). Similarly, for a SRA \( A \), let \( \mathcal{L}(A, v, v') \) denote all the strings \( S = t_1, \cdots, t_n \) such that there exists an accepting run \( [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_n} [n, q_{n+1}, v_{n+1}] \), where \( v_1 = v \) and \( v_{n+1} = v' \).

For every possible SREM \( e \), we will construct a corresponding SRA \( A \) and then prove either that \( \mathcal{L}(e) = \mathcal{L}(A) \) or that \( \mathcal{L}(e, v, v') = \mathcal{L}(A, v, v') \). The latter implies that \( \mathcal{L}(e, \emptyset, v'') = \mathcal{L}(A, \emptyset, v'') \) for some valuation \( v'' \) or equivalently \( \mathcal{L}(e) = \mathcal{L}(A) \), 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 holds 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 \( \mathcal{L}(e_1, v, v'') = \mathcal{L}(A_1, v, v'') \) and that \( \mathcal{L}(e_2, v''', v') = \mathcal{L}(A_2, v''', v') \).

We must be careful, however, with the valuations. If, for example, \( v \) applies to the SRA \( A \), does it also apply to the sub-automaton \( A_1 \), if \( A \) and \( A_1 \) 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 \( \text{reg}(e) \) be a function applied to a SREM \( e \). We define it as follows:

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

For our proofs that follow, we first apply this function to the top expression \( e \) to obtain \( R_{\text{top}} = \text{reg}(e) \) and we use \( R_{\text{top}} \) 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 \( R_{\text{top}} \).

**Assume** \( e := \emptyset \). In this case we know that \( \mathcal{L}(e, v, v') = \emptyset \) for any valuations \( v \) and \( v' \). Thus \( \mathcal{L}(e) = \emptyset \). We can then construct a SRA \( A = (Q, q_s, Q_f, R, \Delta) \) where \( Q = \{q_s\}, Q_f = \emptyset, R = R_{\text{top}} \) and \( \Delta = \emptyset \). It is obvious that \( A \) does not accept any strings. Thus \( \mathcal{L}(A) = \emptyset \).

**Assume** \( e := \epsilon \). We know that \( \mathcal{L}(e) = \{\epsilon\} \). We can then construct a SRA \( A = (Q, q_s, Q_f, R, \Delta) \) where \( Q = \{q_s, q_f\}, Q_f = \{q_f\}, R = R_{\text{top}}, \Delta = \{\delta\} \) and \( \delta = q_s, \epsilon \downarrow \emptyset \rightarrow q_f \). See Figure 7a. It is obvious that \( A \) accepts only the empty string since there is only one path that leads to the final state and this path goes through an \( \epsilon \) transition. Thus \( \mathcal{L}(A) = \{\epsilon\} \).

**Assume** \( e := \phi = R(x_1, \cdots, x_n) \), where \( \phi \) is a condition and all \( x_i \) belong to a set of register variables \( \{r_1, \cdots, r_k\} \). We construct the following SRA \( A = (Q, q_s, Q_f, R, \Delta) \), where \( Q = \{q_s, q_f\}, Q_f = \{q_f\}, R = R_{\text{top}}, \Delta = \{\delta\} \) and \( \delta = q_s, \phi \downarrow \emptyset \rightarrow q_f \). See Figure 7b.

We first prove \( S \in \mathcal{L}(e, v, v') \Rightarrow S \in \mathcal{L}(A, v, v') \) for a string \( S \). It is obvious that \( S \) must be composed of a single element, i.e., \( S = t_1 \). Since \( S = t_1 \) is accepted by \( e \) starting from the valuation \( v \), this means that \( (\phi, S, v) \vdash v' \), with \( v' = v \), according to the second case of Definition 3.13. Thus \( (t_1, v) \vdash \phi \). This then implies that the second case in the definition of a successor configuration (see
(a) Base case of a single $\epsilon$ condition, $e := \epsilon$.
(b) Base case of a single condition, $e := \phi = R(x_1, \cdots, x_n)$.
(c) Base case of a single condition with a write register, $e := \phi \downarrow W = R(x_1, \cdots, x_n) \downarrow \{w\}$.
(d) Concatenation. $e = e_1 \cdot e_2$.
(e) OR. $e = e_1 + e_2$.
(f) Iteration. $e' = e^\ast$.

Figure 7: The cases for constructing a SRA from a SREM.
Definition [4.3] holds. As a result, $A$, upon reading $S$, moves to its final state $q_f$ and accepts $S$. This move does not change the valuation, thus $v' = v$. We have thus proven that $S \in \mathcal{L}(A, v, v')$.

The inverse direction, $S \in \mathcal{L}(A, v, v') \Rightarrow S \in \mathcal{L}(e, v, v')$, can be proven in a similar manner.

Assume $e := \phi = R(x_1, \ldots, x_n) \downarrow w$, where $\phi$ is a condition, all $x_i$ belong to a set of register variables $\{r_1, \ldots, r_k\}$ and $w$ a write register (not necessarily one of $r_i$). We construct the following $SRA$ $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \phi \downarrow \{w\} \rightarrow q_f$. See Figure 7c.

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 4.3). 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 SREM. We first construct $A_1$ and $A_2$, the SRA for $e_1$ and $e_2$ respectively. We construct the following $SRA$ $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = A_1, Q \cup A_2, Q$, $q_s = A_1, q_s$, $Q_f = \{A_2, q_f\}$, $R = R_{top}$, $\Delta = A_1, \Delta \cup A_2, \Delta \cup \{\delta\}$ and $\delta = A_1, q_f, \epsilon \rightarrow A_2, q_s$. See Figure 7d. We thus simply connect $A_1$ and $A_2$ with an $\epsilon$ transition. Notice that $A_1.R$ and $A_2.R$ may overlap. Their union retains only one copy of each register, if a register appears in both of them.

We first prove $S \in \mathcal{L}(e, v, v') \Rightarrow S \in \mathcal{L}(A, v, v')$ for a string $S$. Since $S \in \mathcal{L}(e, v, v')$, $S$ can be broken into two sub-strings $S_1$ and $S_2$ such that $S = S_1 \cdot S_2$, $(e_1, S_1, v) \vdash v''$ and $(e_2, S_2, v'') \vdash v'$. This is equivalent to $S_1 \in \mathcal{L}(e_1, v, v'')$ and $S_2 \in \mathcal{L}(e_2, v'', v')$. From the induction hypothesis (i.e., that what we want to prove holds for the sub-expressions $e_1, e_2$ and their automata $A_1, A_2$) it follows that $S_1 \in \mathcal{L}(A_1, v, v'')$ and $S_2 \in \mathcal{L}(A_2, v'', v')$. Notice that if $A_1$ and $A_2$ have different sets of registers, we can always expand $A_1.R$ and $A_2.R$ 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 $S_1 \in \mathcal{L}(A_1, v, v'')$ it follows that there exists an accepting run $q_1$ of $A_1$ over $S_1$ such that $q_1 = [1, A_1, q_s, v] \rightarrow \cdots \rightarrow [l_1 + 1, A_1, q_f, v'']$. Similarly, from $S_2 \in \mathcal{L}(A_2, v'', v')$ it follows that there exists an accepting run $q_2$ of $A_2$ over $S_2$ such that $q_2 = [1, A_2, q_s, v''] \rightarrow \cdots \rightarrow [l_2 + 1, A_2, q_f, v']$. Let’s construct a run by connecting $q_1$ and $q_2$ with an $\epsilon$ transition: $q = [1, A_1, q_s, v] \rightarrow \cdots \rightarrow [l_1 + 1, A_1, q_f, v''] \Rightarrow A_1, q_f, \epsilon \rightarrow A_2, q_s \Rightarrow [l_1 + 2, A_2, q_s, v''] \rightarrow \cdots \rightarrow [l_1 + l_2 + 1, A_2, q_f, v']$. We can see that this is indeed an accepting run of $A$. Thus $S \in \mathcal{L}(A, v, v')$.

The inverse direction, $S \in \mathcal{L}(A, v, v') \Rightarrow S \in \mathcal{L}(e, v, v')$, can be proven in a similar manner. Since $S \in \mathcal{L}(A, v, v')$, there exists an accepting run $\varrho$ of $A$ over $S$. By the construction of $A$, however, this run must be in the form $\varrho = q_1 \rightarrow q_2$ with $q_1$ being an accepting run of $A_1$ over a string $S_1$ and $q_2$ an accepting run of $A_2$ over $S_2$, where $S = S_1 \cdot S_2$. We then use the induction hypothesis to prove that $S_1 \in \mathcal{L}(e_1, v, v'')$ and $S_2 \in \mathcal{L}(e_2, v'', v')$ and finally that $S \in \mathcal{L}(e, v, v')$.

Assume $e := e_1 + e_2$, where $e_1$ and $e_2$ are SREM. We first construct $A_1$ and $A_2$, the SRA for $e_1$ and $e_2$ respectively. We construct the following $SRA$ $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = A_1, Q \cup A_2, Q \cup \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = A_1, \Delta \cup A_2, \Delta \cup \{\delta_{s,1}, \delta_{s,2}, \delta_{1,f}, \delta_{2,f}\}$ and $\delta_{s,1} = q_s, \epsilon \rightarrow A_1, q_s$, $\delta_{s,2} = q_s, \epsilon \rightarrow A_2, q_s$, $\delta_{1,f} = A_1, q_f, \epsilon \rightarrow q_f$, $\delta_{2,f} = A_2, q_f, \epsilon \rightarrow q_f$. See Figure 7c. We thus create a new state, $q_s$, acting as the start state and connect it through $\epsilon$ transitions to the start states of $A_1$ and $A_2$. We also create a new final state and connect to it the final states of $A_1$ and $A_2$. Again, $A_1.R$ and $A_2.R$ 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 $S \in \mathcal{L}(e, v, v') \Rightarrow S \in \mathcal{L}(A, v, v')$ for a string $S$. If $(e_1, S, v) \vdash v'$, this
implies that \(e_1\) is accepted by \(A_1\). It is thus also accepted by \(A\). Similarly if \((e_2, S, v) \vdash v'\) for \(A_2\). The inverse direction has a similar proof.

**Assume** \(e' := e^*\), where \(e\) is a SREM. We construct a new SRA \(A'\) as shown in Figure 7f. We first prove the SRA for \(e, A\). 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 \(S \in \mathcal{L}(e, v, v') \Rightarrow S \in \mathcal{L}(A, v, v')\) for a string \(S\). Since \(S \in \mathcal{L}(e, v, v')\), \(S = S_1 \cdot S'\) such that \((e, S_1, v) \vdash v''\) and \((e^*, S', v'') \vdash v'\). Equivalently, this implies that \((e, S_1, v) \vdash v_1\) and \((e, S_2, v_1) \vdash v_2\) and \((e, S_3, v_2) \vdash v_3\) etc until \((e, S_n, v_{n-1}) \vdash v_n\), where \(v_n = v'\). We can then construct the run \(q = q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n\). It is easy to see that \(q\) is an accepting run of \(A'\). Similarly for the inverse direction. \(\square\)

### 10.2. Proof of Lemma 5.3

**Lemma 10.2.** For every SRA \(A\) with \(\epsilon\) transitions there exists an equivalent SRA \(A_{\bar{\epsilon}}\) without \(\epsilon\) transitions, i.e., a SRA such that \(\mathcal{L}(A_{\epsilon}) = \mathcal{L}(A_{\bar{\epsilon}})\).

**Proof:**

We first give the algorithm. See Algorithm 2. Note that in this algorithm, the function \(\text{Enclose}\) is the usual function for \(\epsilon\)-enclosure in standard automata theory and we will not repeat it here (see [52]). 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 \(\epsilon\)-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 \(S \in \mathcal{L}(A_{\epsilon}) \iff S \in \mathcal{L}(A_{\bar{\epsilon}})\) for a string \(S\).

We first prove the direction \(S \in \mathcal{L}(A_{\epsilon}) \Rightarrow S \in \mathcal{L}(A_{\bar{\epsilon}})\). The other direction can be proven similarly. Let \(q_\epsilon\) denote an accepting run of \(A_{\epsilon}\) over \(S\), where \(k = |S|\) is the length of \(S\).

\[ q_\epsilon = [1, q_{\epsilon,1} = q_{\epsilon,s}, v_{\epsilon,1} = \#] \xrightarrow{\delta_{\epsilon,1}} [\cdots] \xrightarrow{\delta_{\epsilon,1}} \cdots \text{ sub-run 1} \]

\[ \delta_{\epsilon,2} \rightarrow [2, q_{\epsilon,2}, v_{\epsilon,2}] \xrightarrow{\delta_{\epsilon,2}} [\cdots] \xrightarrow{\delta_{\epsilon,2}} \cdots \text{ sub-run 2} \]

\[ \cdots \]

\[ \delta_{\epsilon,i} \rightarrow [i, q_{\epsilon,i}, v_{\epsilon,i}] \xrightarrow{\delta_{\epsilon,i}} [\cdots] \xrightarrow{\delta_{\epsilon,i}} [i', q_{\epsilon,i'}, v_{\epsilon,i'}] \text{ sub-run i} \] (7)

\[ \delta_{\epsilon,i+1} \rightarrow [i+1, q_{\epsilon,i+1}, v_{\epsilon,i+1}] \xrightarrow{\delta_{\epsilon,i+1}} [\cdots] \xrightarrow{\delta_{\epsilon,i+1}} \cdots \text{ sub-run i+1} \]

\[ \cdots \]

\[ \delta_{\epsilon,k} \rightarrow [k+1, q_{\epsilon,k+1} \in Q_{\epsilon,f}, v_{\epsilon,k+1}] \text{ sub-run k+1} \]
Algorithm 2: Eliminating ε-transitions (EliminateEpsilon).

**Input:** SRA $A_\epsilon$, possibly with ε transitions

**Output:** SRA $A_\bar{\epsilon}$ without ε-transitions

1. $q_{\bar{\epsilon},s} \leftarrow \text{Enclose}(A_\epsilon.q_s); Q_{\bar{\epsilon}} \leftarrow \{q_{\bar{\epsilon},s}\}; \Delta_{\bar{\epsilon}} \leftarrow \emptyset$;
2. **if** $\exists q \in q_{\bar{\epsilon},s} : q \in A_\epsilon.Q_f$ **then**
   3. $Q_{\bar{\epsilon},f} \leftarrow \{q_{\bar{\epsilon},s}\}$;
3. **else**
   4. $Q_{\bar{\epsilon},f} \leftarrow \emptyset$;
   5. frontier $\leftarrow \{q_{\bar{\epsilon},s}\}$;
   6. **foreach** $q_{\bar{\epsilon}} \in$ frontier
      7. **foreach** $q_\epsilon \in q_{\bar{\epsilon}}$ do
         8. **foreach** $\delta_\epsilon \in A_\epsilon.\Delta : \delta_\epsilon.source = q_\epsilon \land \delta_\epsilon \neq \epsilon$ do
            9. $p_\epsilon \leftarrow \delta_\epsilon.target$;
            10. $p_{\bar{\epsilon}} \leftarrow \text{Enclose}(p_\epsilon)$;
            11. $Q_{\bar{\epsilon}} \leftarrow Q_{\bar{\epsilon}} \cup \{p_{\bar{\epsilon}}\}$;
            12. **if** $\exists q \in p_{\bar{\epsilon}}: q \in A_\epsilon.Q_f$ **then**
               13. $Q_{\bar{\epsilon},f} \leftarrow Q_{\bar{\epsilon},f} \cup \{p_{\bar{\epsilon}}\}$;
               14. $\Delta_{\bar{\epsilon}} \leftarrow \Delta_{\bar{\epsilon}} \cup \{\delta_{\bar{\epsilon}}\}$;
               15. frontier $\leftarrow$ frontier $\cup \{p_{\bar{\epsilon}}\}$;
            16. frontier $\leftarrow$ frontier $\setminus \{q_{\bar{\epsilon}}\}$;
         17. $A_{\bar{\epsilon}} \leftarrow (Q_{\bar{\epsilon}}, q_{\bar{\epsilon},s}, Q_{\bar{\epsilon},f}, A_\epsilon.R, \Delta_{\bar{\epsilon}})$;
18. **return** $A_{\bar{\epsilon}}$;
Let $q_\xi$ denote a run of $A_\xi$ over $S$.

$$q_\xi = [1, q_{\xi,1} = q_{\xi,*}, v_{\xi,1} = \emptyset] \quad \text{sub-run 1}$$

$$\delta_{\xi} \rightarrow [2, q_{\xi,2}, v_{\xi,2}] \quad \text{sub-run 2}$$

$$\ldots$$

$$\delta_{\xi,i-1} \rightarrow [i, q_{\xi,i}, v_{\xi,i}] \quad \text{sub-run } i \quad (8)$$

$$\delta_{\xi,i} \rightarrow [i + 1, q_{\xi,i+1}, v_{\xi,i+1}] \quad \text{sub-run } i+1$$

$$\ldots$$

$$\delta_{\xi,k} \rightarrow [k + 1, q_{\xi,k+1}, v_{\xi,k+1}] \quad \text{sub-run } k+1$$

$q_\xi$ necessarily follows $k$ transitions, since it does not have any $\epsilon$-transitions. On the other hand, $q_\xi$ may follow more than $k$ transitions ($j \geq k$), because several $\epsilon$ transitions may intervene between “actual”, non-$\epsilon$ transitions, as shown in Run 7. The number of non-$\epsilon$ transitions is still $k$. $q_\epsilon$ is thus necessarily composed of $k$ “sub-runs”, where the first configuration of each sub-run is reached via a non-$\epsilon$ transition, followed by a sequence of 0 or more $\epsilon$ transitions. Each line in Run 7 is such a sub-run. We can also split $q_\xi$ in sub-runs, but in this case each such sub-run will be simply composed of a single configuration. See Run 8.

We will prove the following. For each sub-run $i$ of $q_\epsilon$, it holds that:

1. $q_{\xi,i} \in q_\epsilon,i$. In fact, $q_{\xi,i} = \text{Enclosure}(q_{\epsilon,i})$.

2. $v_{\epsilon,i} = v_{\xi,i}$, i.e., $A_\epsilon$ and $A_\xi$ have the same register contents at each $i$.

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 $i = k + 1$ as well. Thus, $q_{\xi,k+1} \in Q_{\xi,f,i}$ and $q_\xi$ is an accepting run as well.

First, notice that in each sub-run $i$ of $q_\epsilon$, $v_{\epsilon,i}$ remains the same, since $\epsilon$ transitions never modify the contents of the registers. Thus, in $q_\epsilon$, $v_{\epsilon,i'} = v_{\epsilon,i}$. It is also obviously true that $i' = i$, since $\epsilon$ 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 7 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 $\delta_{\phi,i}$, 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 $\epsilon$ transitions. From the induction hypothesis we know that $q_{\xi,i}$ must be the enclosure of $q_{\epsilon,i}$. From the construction algorithm for $A_\xi$ (Algorithm 2) we also know that the transition $\delta_{\epsilon,i}$ also exists in $A_\xi$, with $q_{\xi,i}$ its source. $\delta_{\xi,i}$ is has the same condition and references the same registers as $\delta_{\epsilon,i}$. Since $\delta_{\epsilon,i}$ is triggered with $v_{\epsilon,i'}$, $\delta_{\xi,i}$ must also be triggered because $v_{\xi} = v_{\epsilon,i}$ (by the induction hypothesis) and thus $v_{\xi} = v_{\epsilon,i'}$. From the construction algorithm, we can see that $q_{\xi,i+1}$ will be the enclosure of $q_{\epsilon,i+1}$. The state $q_\epsilon$ in Algorithm 2 is $q_{\epsilon,i'}$ in Run 7 while state $p_\epsilon$ in the algorithm is state $q_{\epsilon,i+1}$. $p_\xi$ is the thus the enclosure of $q_{\epsilon,i+1}$. Thus there exists $q_{\xi,i+1}$ which we can reach from $q_{\xi,i}$ and which is the enclosure of $q_{\epsilon,i}$. The second part of the induction hypothesis is obviously true.
Lemma 10.3. For every multi-register SRA $A_{mr}$ there exists an equivalent single-register SRA $A_{sr}$, i.e., a single-register SRA such that $\mathcal{L}(A_{mr}) = \mathcal{L}(A_{sr})$.

Proof:
The proof is constructive. We construct a new single-register SRA and show that it has the same language as the multi-register SRA. The new SRA, $A_{sr}$, has the same number of registers as $A_{mr}$. The main difference is that $A_{sr}$ has more states than $A_{mr}$. See [8] for a similar proof about register automata.

Let $A_{mr} = (Q_{mr}, q_{mr,s}, Q_{mr,f}, R_{mr}, \Delta_{mr})$ and $A_{sr} = (Q_{sr}, q_{sr,s}, Q_{sr,f}, R_{sr}, \Delta_{sr})$ denote the multi- and single-register SRA respectively. Let $w = |R_{mr}| = |R_{sr}|$ denote the number of registers (the same for $A_{mr}$ and $A_{sr}$). Let $p = (p_1, \ldots, p_w) \in (2^{R_{mr}})^w : \bigcup_{k=1}^{w} R_{mr} = R_{mr}$ and $p_i \cap p_j = \emptyset$ for $i \neq j$. In other words, each $p_i$ is a subset of $R_{mr}$ and the union of all $p_i$ gives us $R_{mr}$. Therefore, $p$ denotes a partition of $R_{mr}$. For example, if $R_{mr} = \{r_1, r_2, r_3, r_4\}$, a possible partition would be $p = (\{r_1, r_3\}, \{r_2\}, \{r_4\}, \emptyset)$. Let $P_{R_{mr}} = \{p \mid p$ is a partition of $R_{mr}\}$ denote the set of all possible partitions of $R_{mr}$.

The general idea is that we want to establish a correspondence between the registers of $A_{mr}$ and those of $A_{sr}$. If all the registers in $R_{mr}$ have different contents, then each one of them may correspond to a unique register in $R_{sr}$. However, since a transition in $A_{mr}$ may write to multiple registers, at some point in a run of $A_{mr}$, some of its registers will have the same contents. For example, if $R_{mr} = \{r_1, r_2, r_3, r_4\}$, a transition may write to $r_1$ and $r_3$ at the same time. In this case then, the registers of $R_{mr}$ may be partitioned as follows, according to which of them have the same contents: $p = (\{r_1, r_3\}, \{r_2\}, \{r_4\}, \emptyset)$. Now, we could map each register of $R_{rs}$ to one of the $p_i$ in $p$. Repeated values in $R_{mr}$ would then exist as single values in $R_{rs}$. The next issue would then be how we could actually track in a run of $A_{mr}$ the registers that have the same value(s). We could actually achieve this by combining the states of $A_{mr}$ with every possible partition.

For $A_{sr}$, we would then have:

- $Q_{sr} = Q_{mr} \times P_{R_{mr}}$, where \( \times \) indicates the Cartesian product.
- $q_{sr,s} = (q_{mr,s}, p_s)$, where $p_s = (R_{mr}, \emptyset, \ldots, \emptyset)$.
- $Q_{sr,f} = Q_{mr,f} \times P_{R_{mr}}$.
- $R_{mr} = \{r_{mr,1}, \ldots, r_{mr,w}\}$.
- The set of transitions $\Delta_{sr}$ is defined as follows:
We want to show that \( L(A_{mr}) = L(A_{sr}) \). First, assume that \( S \in L(A_{mr}) \) for a string \( S \). We will show that \( S \in L(A_{sr}) \). Let \( q_{mr} \) be an accepting run of \( A_{mr} \) over \( S \):

\[
q_{mr} = [1, q_{mr,1} = q_{mr,s}, v_{mr,1} = \#] \delta_{mr,1}^\rightarrow \\
\ldots \\
\delta_{mr,i-1} \rightarrow [i, q_{mr,i}, v_{mr,i}] \delta_{mr,i}^\rightarrow \\
\ldots \\
\delta_{mr,l} \rightarrow [l + 1, q_{mr,l+1} \in Q_{mr,f}, v_{mr,l+1}]
\]

Let \( q_{sr} \) be a run of \( A_{sr} \) over \( S \):

\[
q_{sr} = [1, q_{sr,1} = q_{sr,s}, v_{sr,1} = \#] \delta_{sr,1}^\rightarrow \\
\ldots \\
\delta_{sr,i-1} \rightarrow [i, q_{sr,i}, v_{sr,i}] \delta_{sr,i}^\rightarrow \\
\ldots \\
\delta_{sr,l} \rightarrow [l + 1, q_{sr,l+1} \in Q_{sr,f}, v_{sr,l+1}]
\]

We need to show that \( q_{sr,l+1} \in Q_{sr,f} \). We can prove this inductively. As our induction hypothesis, we assume that \( q_{sr,i} = (q, p) \), where
1. \( q = q_{mr,i} \) and

2. \( p = (p_{i,1}, \ldots, p_{i,w}) \) such that for all \( 1 \leq k \leq w \) (i.e., all registers of \( A_{sr} \)) \( v_{sr,i}(r_{sr,k}) = v_{mr,i}(r_{mr,k'}) \) for all \( r_{mr,k'} \in p_{i,k} \).

In other words, at the \( i^{th} \) configuration in \( q_{sr} \), we have reached a state \( q_{mr,i} \). The first element of this state must be \( q_{mr,i} \), i.e., the state at the \( i^{th} \) configuration of \( q_{mr} \). The second element must be a partition (of the registers of \( A_q \)). This state must be \( \delta(r) \) and \( \delta(r) \) is not the case, we can always construct an equivalent condition that references all registers, but does not hold for the \( (i+1)^{th} \) one.

First, assume that \( \delta_{mr,i} = q_{mr,i}, \epsilon \rightarrow q_{mr,i+1} \). Then, from the construction of \( A_{sr} \), we know that there exists \( \delta_{sr} \in \Delta_{sr} \) such that \( \delta_{sr} = (q_{mr,i}, p), \epsilon \rightarrow (q_{mr,i+1}, p) \). The first part of the induction hypothesis then still holds because we can move to a state whose first element is \( q_{mr,i+1} \). The second part of the hypothesis also holds because \( p \) remains the same and we already know that this part holds for \( p \) from the hypothesis itself.

Now, assume that \( \delta_{mr,i} = q_{mr,i} \), \( \phi_{mr} \rightarrow q_{mr,i+1} \), with \( \phi_{mr} \neq \epsilon \). Then, from the construction of \( A_{sr} \), we know that there exists \( \delta_{sr} \in \Delta_{sr} \) such that \( \delta_{sr} = (q_{mr,i}, p), \phi_{sr} \rightarrow (q_{mr,i+1}, p) \) and the condition \( \phi_{sr} \) is triggered. We can prove the latter claim about \( \phi_{sr} \) by noticing that \( \phi_{sr} \) is triggered. Now, from the construction, \( \phi_{sr} \) is the same as \( \phi_{mr} \) with its arguments/registers appropriately replaced, as described above. Without loss of generality, assume that \( \phi_{mr} \) references all of its registers (if this is not the case, we can always construct an equivalent condition that references all registers, but does not actually access any of the redundant ones). We can write it as follows:

\[
\phi_{mr} = \phi(r_{mr,1}, \ldots, r_{mr,w})
\]

\( \phi_{sr} \) can be written as follows:

\[
\phi_{sr} = \phi(r_{sr,i_1}, \ldots, r_{mr,i_w})
\]

where \( i_1 \) is such that \( r_{mr,1} \in p_{i_1} \), i.e., \( i_1 \) is the partition set from \( p \) where \( r_{mr,1} \) belongs. Similarly for \( i_2 \), etc. For example, if \( R_{mr} = \{r_{mr,1}, r_{mr,2}, r_{mr,3}, r_{mr,4} \} \) and \( p = (p_1, p_2, p_3, p_4) = (\{r_{mr,1}, r_{mr,3} \}, \{r_{mr,2} \}, \{r_{mr,4} \}, \emptyset) \) then

\[
\phi_{mr} = \phi(r_{mr,1}, r_{mr,2}, r_{mr,3}, r_{mr,4})
\]

and

\[
\phi_{sr} = \phi(r_{sr,1}, r_{sr,2}, r_{sr,1}, r_{sr,3})
\]

From the induction hypothesis, we know, however, that \( v_{mr,i}(r_{mr,i}) = v_{sr,i}(r_{sr,i}) \). Therefore, \( \phi_{mr} \) and \( \phi_{sr} \) essentially have the same arguments. Since \( \phi_{sr} \) is also triggered, the first part of the induction hypothesis holds for \( (i+1) \) as well. The second part also holds since \( p \) again remains the same.

Finally, assume that \( \delta_{mr,i} = q_{mr,i}, \phi_{mr} \downarrow R_w \rightarrow q_{mr,i+1} \), with \( \phi_{mr} \neq \epsilon \) and \( R_w \neq \emptyset \). Then, from the construction of \( A_{sr} \), we know that there exists \( \delta_{sr} \in \Delta_{sr} \) such that \( \delta_{sr} = (q_{mr,i}, p), \phi_{sr} \downarrow r_{sr,k} \rightarrow \).
(qmr,i+1, p′) and the condition ϕsr is triggered. The proof for ϕsr being triggered is the same as in the previous case. We additionally need to prove the second part of the induction hypothesis, since p now becomes p′. In other words, we need to prove for p′ that the contents of a register j of Asr are the same as those of the registers of Amr contained in the jth set in the partition p′. Indeed, this is the case. From the construction, we know that the partition set pk (reminder: rsrk is the write register in δsr) becomes p′k = pk ∪ Rw. Since pk ⊂ Rw, this means that p′k = Rw. Thus the hypothesis still holds for p′k, since rsrk and all registers in Rw will have the same value. The hypothesis also holds for k′ ≠ k. Since p′k′ = pk′ \ Rw and the hypothesis holds for p, this means that pk′ had registers with the same contents before the writing to Rw. If we remove from pk′ the changed registers to obtain p′k′, then p′k′ will still have registers with the same contents after the writing to Rw. Additionally, since rsrk′ was not changed, it will still have the same contents as the registers in p′k′.

We know that the hypothesis holds for i = 1 in the runs qmr and qsr, because qsr,s = (qmr,s, ps), with ps = (Rmr,…, 0). The first part of the hypothesis obviously holds. The second part also holds since all registers are empty in both runs at the beginning. Therefore, the hypothesis holds for i = 2, i = 3, etc. qsr is thus an accepting run of Asr over S.

We have proven that S ∈ L(Amr) ⇒ S ∈ L(Asr). The inverse direction, i.e., S ∈ L(Asr) ⇒ S ∈ L(Amr), can be proven similarly. We first assume that there is an accepting run qsr of Asr over S and then show, in a similar manner, that there exists an accepting runf = qmr of Amr over S. □

10.4. Proof of Theorem 5.6

Theorem 10.4. For every SRA A there exists an equivalent SREM e, i.e., a SREM such that L(A) = L(e).

Proof:

The proof develops along lines similar to the corresponding proof for classical and register automata [53][11]. It uses a generalized version of SRA, denoted by gSRA. These are SRA whose transitions are not equipped with a single condition, but with a whole SREM. For example, the gSRA A = (Q, qs, Qf, R, Δ), where Q = {qs, qf}, Qf = {qf}, R = {r1}, Δ = {δ} and δ = qs, (ϕ1 ↓ {r1}) · (ϕ2(r1)) → qf. The single transition δ can read two characters at the same time, apply ϕ1 on the first one, store this character in r1 and then apply ϕ2 on the second character and the contents of r1. We assume that all ε transitions have been eliminated and that the SRA is single-register (and if not, it has been converted to one, as shown in Appendix 10.3). We also demand that a) gSRA have a single start state with no incoming transitions and with outgoing transitions to every other state, b) they have a single final state with no outgoing transitions and with incoming transitions from every other state, c) there is an arrow connecting any two other states. We say that a gSRA Ag accepts a string S if S = S1 · S2 · · · · S k and there exists a run g = [1, q1, v1] → · · · · [i, qi, vi] → · · · → [l + 1, qk+1, vk+1], where the state of the first configuration is the start state of Ag (q1 = Ag.qs), the state of the last configuration is its final state (qk+1 = Ag.qf) and for each i there exists a transition δ of Ag such that δ = qi, ei → qi+1 and Si ∈ L(ei, vi, vi+1).
We first convert the initial \textit{SRA} \(A\) to a \textit{gSRA} \(A_g\) as follows. We add a new start state and connect it to the old start state with an \(\epsilon\) transition. We also connect with such a transition the old final state to a new final state. It there are multiple transitions between any two states, we combine them into a single transition whose condition will be the union of the conditions of the previous transitions (we can do this since we are allowed to have SREM on the transitions now). Finally, if there exist states that are not connected to each other, we connect them with \(\emptyset\) transitions (making sure that we do not add incoming transitions to the start state or outgoing transitions to the final state). This procedure will produce a \textit{gSRA} \(A_g\) which will be equivalent in terms of its language to the original \textit{SRA} \(A\).

The basic idea is to start removing states from \(A_g\), one at a time, without affecting the language it accepts. This procedure is repeated until we are left with 2 states. At this point, the \textit{gSRA} will have one start and one final state, connected with a single transition. The \textit{SREM} on this transition is finally returned as the \textit{SREM} corresponding to the initial \textit{SRA} \(A\). The critical step in this process is of course the one where a state is removed. We must ensure that any repairs we make to the remaining transitions do not affect the automaton’s language. We first select a state to remove, \(q_{\text{rip}}\). This can be any state, except for the start or the final states. We then check all pairs of states \(q_i\) and \(q_j\). We need to make sure that the new automaton will be able to move from \(q_i\) to \(q_j\) with exactly the same strings as when \(q_{\text{rip}}\) was present. We thus have to modify the \textit{SREM} on the transition from \(q_i\) to \(q_j\). The modification is the following. Assume, that, in the old automaton, before the removal, \(q_i\) becomes \(q_j\) via Algorithm 3, are equivalent, i.e., that \(S \in \mathcal{L}(A_{g,n}, v, v') \equiv S \in \mathcal{L}(A_{g,n-1}, v, v')\). If we can prove this, then the last step of the recursion of Algorithm 3 (Line 2) will give us an automaton that is equivalent to our initial \textit{SRA} \(A\). Moreover, the \textit{SREM} on the single transition of this \textit{gSRA} is obviously the desired \textit{SREM}.

First, assume that \(S \in \mathcal{L}(A_{g,n}, v, v')\). Then, there exists an accepting run of \(A_{g,n}\)

\[
q = [1, A_{g,n}.q_s, v_1] \rightarrow \cdots \rightarrow [i, q_i, v_i] \rightarrow \cdots \rightarrow [l + 1, A_{g,n}.q_f, v_{k+1}]
\]

Now, assume that \(q_i \neq q_{\text{rip}}\) for all \(q_i\) of \(q\), including, of course, the start and final states. We claim that this run would also be an accepting run for \(A_{g,n-1}\). To prove this, notice that between any two successive configurations \([i, q_i, v_i] \rightarrow [j, q_j, v_j]\) appearing in \(q\), there exists a transition that is triggered between \(q_i\) and \(q_j\). Let \(e_4\) denote the \textit{SREM} on this transition. In \(A_{g,n-1}\) the \textit{SREM} on this transition would become \((e_1 \cdot (e_2)^* \cdot e_3) + e_4\). Thus, it would also be triggered, due to the presence

Notice that such transitions exist for every pair \(q_i\) and \(q_j\), since, by definition, transitions exist between all pairs of states. Then, after removing \(q_{\text{rip}}\), the \textit{SREM} on the transition from \(q_i\) to \(q_j\) becomes \((e_1 \cdot (e_2)^* \cdot e_3) + e_4\). See Algorithm 3.
Algorithm 3: Converting a $gSRA$ with $n$ states to a $gSRA$ with 2 states.

| Input: | A $gSRA\ A$ with $n$ states. |
| Output: | A $gSRA\ A_g$ with 2 states, equivalent to $A$. |
| if $|A.Q| = 2$ then |
| return $A$; |
| else |
| Pick an element $q_{rip}$ from $A.Q$ other than $A.q_s$ or $A.q_f$; |
| $Q' \leftarrow Q - \{q_{rip}\}$; |
| $\Delta' \leftarrow \emptyset$; |
| /* Assume $\delta(q_i, q_j)$ returns the SREM on the transition from $q_i$ to $q_j$. */ |
| foreach $q_i \in Q' - \{A.q_f\}$ and $q_j \in Q' - \{q_f\}$ do |
| $\delta' \leftarrow q_i, ((e_1 \cdot (e_2)^* \cdot e_3) + e_4) \rightarrow q_j$ for $e_1 = \delta(q_i, q_{rip}), e_2 = \delta(q_{rip}, q_{rip})$, $e_3 = \delta(q_{rip}, q_j), e_4 = \delta(q_i, q_j)$; |
| $\Delta' \leftarrow \Delta' \cup \{\delta'\}$; |
| $A' \leftarrow (Q', A.q_s, A.q_f, A.R, \Delta')$; |
| return CONVERT($A'$); |

We claim that, if we remove from $\varrho$ all configurations with $q_{rip}$, then the remaining configurations would form an accepting run of $A_{g,n-1}$. To prove this, notice that the transition from $q_i$ to $q_{rip}$ in $\varrho$ would happen through the $e_1$ SREM. Then, every loop (if any) from $q_{rip}$ to itself would occur because of the $e_2$ SREM. Finally, the jump from $q_{rip}$ to $q_l$ would happen through $e_3$. Thus, the move from $q_i$ to $q_l$ would happen via $e_1 \cdot (e_2)^* \cdot e_3$. But this is exactly one of the disjuncts on the transition from $q_i$ to $q_l$ in $A_{g,n-1}$. We can therefore remove all consecutive configurations with $q_{rip}$ from $\varrho$. If there are multiple such sequences in $\varrho$, we can repeat the same process as many times as necessary.

We have thus proven that $S \in \mathcal{L}(A_{g,n-1}, v, v') \Rightarrow S \in \mathcal{L}(A_{g,n-1}, v, v')$

Conversely, assume that $S \in \mathcal{L}(A_{g,n-1}, v, v')$. There is then an accepting run $\varrho$ of $A_{g,n-1}$. The move between each sequence of successive configurations, from state $q_i$ to $q_j$, happens either due to $e_4$ or due to $e_1 \cdot (e_2)^* \cdot e_3$. In the former case, we can retain this move as is in a new run for $A_{g,n}$. In the latter case, we can insert between the configurations of $q_i$ and $q_j$ a sequence of configurations with $q_{rip}$, as already described previously. Thus, $S \in \mathcal{L}(A_{g,n-1}, v, v') \Rightarrow S \in \mathcal{L}(A_{g,n}, v, v')$. This completes our proof.  

\[ \square \]
10.5. Proof of Theorem 5.9

Theorem 10.5. \( SRA \) and \( SREM \) are closed under union, intersection, concatenation and Kleene-star.

Proof:
For union, concatenation and Kleene-star the proof is essentially the proof for converting \( SREM \) to \( SRA \). For concatenation, if we have \( SRA \) \( A_1 \) and \( A_2 \) we construct \( A \) as in Figure 7d. For union, we construct the \( SRA \) as in Figure 7e. For Kleene-star, we construct the \( SRA \) as in Figure 7f. The only difference in these constructions is that we now assume, without loss of generality, that the \( A_1.R \cap A_2.R = \emptyset \), i.e., that \( A_1 \) and \( A_2 \) have different sets of registers and that the automaton \( A \) constructed from \( A_1 \) and \( A_2 \) retains all registers of both \( A_1 \) and \( A_2 \). For example, if we have two \( SRA \) \( A_1 \) and \( A_2 \) and we want to construct a \( SRA \) \( A \) such that \( \mathcal{L}(A) = \mathcal{L}(A_1) \cdot \mathcal{L}(A_2) \) then we connect \( A \)'s final state to \( A_2 \)'s start state via an \( \epsilon \) transition. It is easy to see that if \( S_1 \in \mathcal{L}(A_1) \) and \( S_2 \in \mathcal{L}(A_2) \) then \( S = S_1 \cdot S_2 \in \mathcal{L}(A) \). \( S_1 \) will force \( A \) to move to \( A_1 \)'s final state (both \( A \) and \( A_1 \) start with empty registers). Subsequently, \( A \) will jump to \( A_2 \)'s start state and then \( S_2 \) will force \( A \) to go to \( A_2 \)'s final state which is \( A \)'s final state, since \( A_2 \)'s registers in \( A \) are empty when \( A_2 \) starts reading \( S_2 \).

We will now prove closure under intersection. Let \( A_1 = (Q_1, q_{1,s}, Q_{1,f}, R_1, \Delta_1) \) and \( A_2 = (Q_2, q_{2,s}, Q_{2,f}, R_2, \Delta_2) \) be two \( SRA \). We want to construct a \( SRA \) \( A = (Q, q_s, Q_f, R, \Delta) \) such that \( \mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \). We construct \( A \) 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 \( R_1 \cup R_2 = \emptyset \).
- For each \( q = (q_1, q_2) \in Q \) we add a transition \( \delta \) to \( q' = (q'_1, q'_2) \in Q \) if there exists a transition \( \delta_1 \) from \( q_1 \) to \( q'_1 \) in \( A_1 \) and a transition \( \delta_2 \) from \( q_2 \) to \( q'_2 \) in \( A_2 \). The condition of \( \delta \) is \( \phi = \delta_1.\phi \land \delta_2.\phi \). The write registers of \( \delta \) 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 \( SRA \), even if \( A_1 \) and \( A_2 \) are single-register). Thus, \( \delta = (q_1, q_2), (\delta_1.\phi \land \delta_2.\phi) \downarrow (\delta_1.W \cup \delta_2.W) \rightarrow (q'_1, q'_2) \).

It is evident that, if a string \( S \) is accepted by both \( A_1 \) and \( A_2 \), it is also accepted by \( A \). If \( A \) is not accepted either by \( A_1 \) or \( A_2 \), then it is not accepted by \( A \). Therefore, \( \mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \).

Since \( SRA \) and \( SREM \) are equivalent, \( SREM \) are also closed under union, intersection, concatenation and Kleene-star. \( \square \)

10.6. Proof of Theorem 5.10

Theorem 10.6. \( SRA \) and \( SREM \) are not closed under complement.

Proof:
The proof is by a counter example. Let \( A \) denote the \( SRA \) of Figure 8. This \( SRA \) 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, \( A \) accepts strings in which there are two elements with the same type, regardless of the length of \( S \). Assume that there exists a \( SRA \) \( A_c \) which accepts only when \( A \) does not accept. In other words, \( A_c \) accepts all strings \( S \) whose elements all have a different type. Let \( k = |A_c.R| \) be the number of registers of \( A_c \). Let \( |S| = k + m \), where \( m > 1 \), be the length of a string \( S \) whose elements all have different types. However, \( A_c \) cannot possibly exist. At the end of \( S \), as \( A_c \) is ready to read the last element of \( S \), it must have stored all of the previous \( k + m - 1 \) elements of \( S \). But \( A \) has only \( k \) registers, whereas \( k + m - 1 > k \), since \( m > 1 \). Thus, \( A_c \) cannot exist.

\[\begin{align*}
\phi(r_1) := & \sim \cdot \text{type} = v(r_1) \cdot \text{type} \\
\phi_1 := & \sim \cdot \text{type} = a \\
\phi_2(r_1) := & \sim \cdot \text{type} = b \land v(r_1) \cdot \text{id} = \sim \cdot \text{id}
\end{align*}\]

Figure 8: \( SRA \) accepting strings which have the same type in two elements. Notice that \( \sim \) denotes the current event (last event read from the string).

\[\begin{align*}
\phi_1 := & \sim \cdot \text{type} = a \\
\phi_2(r_1) := & \sim \cdot \text{type} = b \land v(r_1) \cdot \text{id} = \sim \cdot \text{id}
\end{align*}\]

Figure 9: \( SRA \) accepting all strings containing an \( a \) element followed by a \( b \) element, whose identifiers are the same.

10.7. Proof of Theorem 5.11

**Theorem 10.7.** \( SRA \) are not closed under determinization.

**Proof:**
The proof is again by a counter example. Let \( A \) denote the \( SRA \) of Figure 9. This \( SRA \) reads strings composed of tuples. Each tuple contains an attribute, called \( \text{type} \), taking values from a finite or infinite alphabet. It also contains another tuple, called \( \text{id} \), taking integer values. \( A \) 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 \( dSRA \) \( A_d \) with \( k \) registers which is equivalent to \( A \). Let

\[ S = (a, 1)(b, 2) \]
be a string given to $A_d$. 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 $S'_1 = (a, 1)$, $A_d$ must store it in the register $r_1$, since $A_d$ 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 $S''_1 = (b, 2)$ and $S' = (a, 2)(a, 3)(b, 2)$ would not be accepted. Now, let 

$$S'' = (a, \cdots)(a, \cdots) \cdots (a, \cdots)(b, 2)$$

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 $A_d$ can store at most $k$ different elements. Therefore, there does not exist a $dSRA$ which is equivalent to $A$. \hfill $\Box$

10.8. Structural Properties of SRA

In the proofs that follow, we will need to refer to some structural properties of $SRA$. We present them here. Without loss of generality, we assume that each state of a $SRA$ is accessible from its start state. Inaccessible states can always be removed without affecting the behavior of an automaton. Therefore, a $SRA$, in terms of its structure, can be viewed as a weakly connected directed graph. The usual notions of walks and trails from graph theory also apply for a $SRA$. However, since we are interested in walks and trails from its start state, and in order to avoid introducing new notation and terminology, in what follows, we will stick to the already introduced terms. We will talk about states, instead of nodes/vertices, and about transitions, instead of edges.

**Definition 10.8. (Walk over SRA)**

A walk $w$ over a $SRA$ $A$ is a sequence of transitions $w = \langle \delta_1, \cdots, \delta_k \rangle$, such that:

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

We say that such a walk is of length $k$. By $W_A$ we denote the set of all walks over $A$ and by $W_{A,q}$, we denote the set of walks over $A$ that end in state $q$, i.e., $W_{A,q} = \{ w : w \in W_A \land \delta_k.\text{target} = q \}$.

**Definition 10.9. (Trail over SRA)**

A trail $t$ over a $SRA$ $A$ is a walk $w = \langle \delta_1, \cdots, \delta_k \rangle$ over $A$, such that:

- $\forall \delta_i, \delta_j \delta_i.\text{source} \neq \delta_j.\text{source}$
\[ \forall \delta_i \text{. source } \neq \delta_i \text{. target} \]

If \( T_A \) is the set of all trails over \( A \), \( T_{A,q} \) is the set of all trails ending in state \( q \), i.e., \( T_{A,q} = \{ t : t \in T_A \land \delta_k \text{. target} = q \} \).

In other words, a trail is a walk without state revisits (and, as a consequence, without transition revisits).

**Proposition 10.10. (Every walk contains a trail)**

For every walk to a non-start state \( q \) (\( w = < \delta_1, \cdots, \delta_k > \in W_{A,q}, q \in A.Q \setminus \{ A.q_s \} \)), there exists a trail to \( q \) (\( t = < \delta'_1, \cdots, \delta'_l > \in T_{A,q} \)) such that all transitions of the trail (\( \delta'_i \)) appear in the walk \( w \) in the same order as in the trail, i.e., \( w \) can be written as \( w = < \cdots, \delta'_1, \cdots, \delta'_l, \cdots > \). We say that \( t \) is contained in \( w \).

**Proof:**

The proof is by induction on the length of the walk. The proposition trivially holds for walks of length \( k = 1 \). For walks of length \( k + 1 \), if \( w \) is already a trail, then the proposition holds for \( k + 1 \). If \( w \) is not a trail, then a state is visited at least twice. Removing all transitions between these two visits results in a walk for which, by the induction hypothesis, the proposition already holds. Therefore, it holds for the complete walk too.

We give a detailed proof by induction on the length of the walk.

- **Base case for \( k = 1 \).** If \( w = < \delta_1 > \) is a walk to \( q \), then, \( t = < \delta_1 > \) is also a trail to \( q \), since \( q \neq q^s \).

- **Assume** \( w = q_1 \delta_1 \rightarrow q_2 \cdots q_i \delta_i \rightarrow q_{i+1} \cdots q_j \delta_j \rightarrow q_{j+1} \cdots q_k \delta_k \rightarrow q_{k+1} \delta_{k+1} \rightarrow q_{k+2} \) is a walk of length \( k + 1 \) (where we use a slightly different notation to explicitly show the visited states).

  - If \( w \) is already a trail, then the proposition holds for \( k + 1 \).
  - If \( w \) is not a trail, then a state is visited at least twice. Assume that it is \( q_i \), visited again as \( q_j \), i.e., \( q_i = q_j \). Remove from \( w \) all transitions \( \delta_i \), \( i \leq l < j \). Then we get
    \[ w' = q_1 \delta_1 \rightarrow q_2 \cdots q_i \delta_i \rightarrow q_{j+1} \cdots q_k \delta_k \rightarrow q_{k+1} \delta_{k+1} \rightarrow q_{k+2} \]
    or, equivalently, since \( q_i = q_j \)
    \[ w' = q_1 \delta_1 \rightarrow q_2 \cdots q_j \delta_j \rightarrow q_{j+1} \cdots q_k \delta_k \rightarrow q_{k+1} \delta_{k+1} \rightarrow q_{k+2} \]

Notice that \( w' \) is indeed a walk, since all its transitions are valid, including the one that stitches together the two sub-walks (\( q_j \delta_j \rightarrow q_{j+1} \)). Moreover, its length is at most \( k \), since we removed at least one transition. Therefore, by the induction hypothesis, there exists a trail \( t' \) to \( q_{k+2} \), contained in \( w' \). But \( t' \) is also contained in \( w \). Therefore, \( t' \) is a trail to \( q_{k+2} \) contained in \( w \) and the proposition holds for walks of length \( k + 1 \) as well.
Definition 10.11. (Register appearance in a trail)
We say that a register $r$ appears in a trail if there exists at least one transition $\delta$ in the trail such that $r \in \delta.W$.
In other words, a trail must write to $r$ to say that it appears in it.

Definition 10.12. (Walk induced by a run)
If $\varrho = [1, q_s, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_{n-1}} [n, q_n, v_n]$ is a run of a SRA $A$, then $w_{\varrho} = \langle \delta_1, \cdots, \delta_{n-1} \rangle$ is the walk induced by $\varrho$.

10.9. Proof of Lemma 5.13

Theorem 10.13. For every windowed SREM 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 4 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 $\epsilon$-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. 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 5. 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 $\varrho_e$ is a run of $A_e$ of length up to $w$ over a string $S$ ($\varrho_e$ has at most $w$ transitions), then the corresponding run $\varrho_{e_w}$ of $A_{e_w}$ is indeed a run and if $\varrho_e$ is accepting so is $\varrho_{e_w}$. By definition, since the runs have no $\epsilon$-transitions and are at most of length $w$, $|S| \leq w$.

We first prove the following proposition:
Algorithm 4: Constructing SRA for a windowed SREM (ConstructWSRA).

**Input:** Windowed SREM $e' := e^{[1..w]}

**Output:** SRA $A_{e'}$ equivalent to $e'$

1. $A_{e,e} \leftarrow \text{ConstructSRA}(e)$; // As described in Appendix 10.1.
2. $A_{e,ms} \leftarrow \text{EliminateEpsilon}(A_{e,e})$; // See Algorithm 2. $A_{e,ms}$ might be multi-register.
3. $A_{e} \leftarrow \text{ConvertToSingleRegister}(A_{e,ms})$; // As described in Appendix 10.3.
4. $A_{e'} \leftarrow \text{Unroll}(A_{e,w})$; // See Algorithm 5.
5. return $A_{e'}$;

Algorithm 5: Unrolling cycles for windowed SREM (Unroll).

**Input:** SRA $A$ and integer $k \geq 0$

**Output:** SRA $A_{k}$ with runs of length up to $k$

1. if $k = 0$ then
2. (1) $(A_{k}, \text{Frontier}, \text{CopyOfQ}, \text{CopyOfR}) \leftarrow \text{Unroll0}(A)$; // Algorithm 6
3. else
4. (2) $(A_{k}, \text{Frontier}, \text{CopyOfQ}, \text{CopyOfR}) \leftarrow \text{UnrollK}(A, k)$; // Algorithm 7
5. end
6. return $(A_{k}, \text{Frontier}, \text{CopyOfQ}, \text{CopyOfR})$;

Algorithm 6: Unrolling cycles for windowed SREM, base case (Unroll0).

**Input:** SRA $A$

**Output:** SRA $A_{0}$ with runs of length 0

1. $q \leftarrow \text{CreateNewState}();$
2. $\text{CopyOfQ} \leftarrow \{q \rightarrow A.q_{s}\};$
3. $\text{CopyOfR} \leftarrow \emptyset;$
4. $\text{Frontier} \leftarrow \{q\};$
5. $Q_{f} \leftarrow \emptyset;$
6. if $A.q_{s} \in A.Q_{f}$ then
7. (1) $Q_{f} \leftarrow Q_{f} \cup \{q\};$
8. else
9. $A_{0} \leftarrow (\{q\}, q, Q_{f}, \emptyset, \emptyset);$  
10. return $(A_{0}, \text{Frontier}, \text{CopyOfQ}, \text{CopyOfR})$;
**Algorithm 7:** 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 \( k \)

\[
\begin{align*}
(A_{k-1}, \text{Frontier}, \text{CopyOfQ}, \text{CopyOfR}) &\leftarrow \text{Unroll}(A, k - 1); \\
\text{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; \\
\textbf{foreach} q \in \text{Frontier} \textbf{ do} \\
q_c &\leftarrow \text{CopyOfQ}(q); \\
\textbf{foreach} \delta \in A.\Delta : \delta.\text{source} = q_c \textbf{ do} \\
q_{\text{new}} &\leftarrow \text{CreateNewState}(); \\
Q_k &\leftarrow Q_k \cup \{q_{\text{new}}\}; \\
\text{CopyOfQ} &\leftarrow \text{CopyOfQ} \cup \{q_{\text{new}} \rightarrow \delta.\text{target}\}; \\
\textbf{if} \delta.\text{target} \in A.Q_f \textbf{ then} \\
Q_{k,f} &\leftarrow Q_{k,f} \cup \{q_{\text{new}}\}; \\
\textbf{if} \delta.W = \emptyset \textbf{ then} \\
R_{\text{new}} &\leftarrow \emptyset; \\
\textbf{else} \\
r_{\text{new}} &\leftarrow \text{CreateNewRegister}(); \\
R_k &\leftarrow R_k \cup \{r_{\text{new}}\}; \\
R_{\text{new}} &\leftarrow \{r_{\text{new}}\}; \\
\text{CopyOfR} &\leftarrow \text{CopyOfR} \cup \{r_{\text{new}} \rightarrow \delta.r\}; \quad // \ \delta.r \text{ single element of } \delta.W \\
\phi_{\text{new}} &\leftarrow \delta.\phi; \\
rs_{\text{new}} &\leftarrow \emptyset; \\
\text{/* By } \delta.\phi.rs \text{ we denote the register selection of } \delta.\phi, \text{ i.e., all the registers} \\
\text{referenced by } \delta.\phi \text{ in its arguments. } rs \text{ is represented as a list. */} \\
\textbf{foreach} r \in \delta.\phi.rs \textbf{ do} \\
\text{/* } \text{FindLastAppearance} \text{ returns a register that is a copy of } r \text{ and appears} \\
\text{last in the trail of } A_{k-1} \text{ to } q \text{ (no other copies of } r \text{ appear after} \\
r_{\text{latest}} \text{). Due to the construction, only a single walk/trail to } q \text{ exists.} \\
\text{/* */} \\
r_{\text{latest}} &\leftarrow \text{FindLastAppearance}(r, q, A_{k-1}); \\
\text{/* } :: \text{ denotes the operation of appending an element at the end of a list.} \\
\text{r_{\text{latest}} is appended at the end of } rs_{\text{new}}. \quad */ \\
rs_{\text{new}} &\leftarrow rs_{\text{new}} :: r_{\text{latest}}; \\
\delta_{\text{new}} &\leftarrow \text{CreateNewTransition}(q, \phi_{\text{new}}(rs_{\text{new}}) \downarrow R_{\text{new}} \rightarrow q_{\text{new}}); \\
\Delta_k &\leftarrow \Delta_k \cup \{\delta_{\text{new}}\}; \\
\text{NextFrontier} &\leftarrow \text{NextFrontier} \cup \{q_{\text{new}}\}; \\
A_k &\leftarrow (Q_k, A_{k-1}.q_s, Q_{k,f}, R_k, \Delta_k); \\
\text{return } (A_k, \text{NextFrontier}, \text{CopyOfQ}, \text{CopyOfR});
\end{align*}
\]
Proposition 10.14. There exists a run of $A_e$ over a string $S$ of length up to $w$

$$q_e = [1, q_{e,1} = A_e.q_s, v_{e,1}] \xrightarrow{\delta_{e,1}} \cdots \xrightarrow{\delta_{e,n-1}} [n, q_{e,n}, v_{e,n}]$$

iff there exists a run $q_{ew}$ of $A_{ew}$

$$q_{ew} = [1, q_{ew,1} = A_{ew}.q_s, v_{ew,1}] \xrightarrow{\delta_{ew,1}} \cdots \xrightarrow{\delta_{ew,n-1}} [n, q_{ew,n}, v_{ew,n}]$$

such that:

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

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 $r_{ew}$, where $CopyOfR(r_{ew}) = r_e$, appears last if no other copies of $r_e$ appear after $r_{ew}$ in a run. The notion of a register’s (last) appearance also applies for walks of $A_{ew}$, since $A_{ew}$ is a directed acyclic graph, as can be seen by Algorithms 6 and 7 (they always expand “forward” the SRA, without creating any cycles and without converging 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 $q_e$ 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, $v_{e,i} = v_{ew,i} = \#$ for all registers. By Algorithm 6(line 2), we know that $CopyOf(q_{ew,0}) = q_{e,0}$.

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

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

and

$$q_{ew,k+1} = \cdots [k, q_{ew,k}, v_{ew,k}] \xrightarrow{\delta_{ew,k}} [k+1, q_{ew,k+1}, v_{ew,k+1}]$$

be the runs of $A_e$ and $A_{ew}$ respectively of length $k + 1$ over the same $k + 1$ elements of a string $S$. We know that $q_{e,k+1}$ is an actual run and we need to construct $q_{ew,k+1}$, knowing, by the induction hypothesis, that there is an actual run up to $q_{ew,i+k}$. 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 $\delta_{ew,k}$ with the same condition from $q_{ew,k}$ to $q_{ew,k+1}$ such that $CopyOfQ(q_{ew,k+1}) = q_{e,k+1}$. Moreover, if $\delta_{e,k}$ is triggered, so does $\delta_{ew,k}$, because the registers in the register selection of $\delta_{ew,k}$ 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 $q_{ew}$ that appear last. But these are exactly the registers in $\delta_{ew,k}.\phi.rs$ (see line 22 in Algorithm 7). We can also see that the part of the proposition concerning the valuations $v$ also holds. If $\delta_{e,k}.W = \{r_e\}$ and $\delta_{ew,k}.W = \{r_{ew}\}$, then we know, by the construction algorithm (line 18), that $CopyOfR(r_{ew}) = r_e$ and $r_{ew}$ will be the last appearance of a copy of $r_e$ in $q_{ew,k+1}$. Thus the proposition holds for $0 < k + 1 \leq w$ as well. \qed
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.

10.10. Proof of Theorem 5.15

Theorem 10.15. For every windowed SREM there exists an equivalent deterministic SRA.

Proof:
The process for constructing a deterministic SRA ($dSRA$) from a windowed SREM is shown in Algorithm 8. 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 10.7 where we showed that SRA are not in general determinizable.

First, we will prove the following proposition:

Proposition 10.16. There exists a run $\varrho_n$ over a string $S$ which $A_n$ can follow by reading the first $k$ tuples of $S$, iff there exists a run $\varrho_d$ that $A_d$ can follow by reading the same first $k$ tuples, such that, if

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

and

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

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 $\varrho_n$, then it appears in $\varrho_d$
- $v_{n,i}(r) = v_{d,i}(r)$ for every $r$ that appears in $\varrho_n$ (and $\varrho_d$)
Algorithm 8: Determinization.

Input: Windowed \( SREM \) \( e' := e^{1..n} \)

Output: Deterministic \( SRA \) \( A_d \) equivalent to \( e' \)

1. \( A_n \leftarrow \text{ConstructWSRA}(e') \); // See Algorithm 4
2. \( Q_d \leftarrow \text{ConstructPowerSet}(A_n.Q) \);
3. \( \Delta_d \leftarrow \emptyset; Q_{f,d} \leftarrow \emptyset \);
4. foreach \( q_d \in Q_d \) do
   5. if \( q_d \cap A_n.Q_f \neq \emptyset \) then
      6. \( Q_{f,d} \leftarrow Q_{f,d} \cup \{q_d\} \);
   7. \( \text{Conditions} \leftarrow () \); \( rs_d \leftarrow () \);
5. foreach \( q_n \in q_d \) do
   6. foreach \( \delta_n \in A_n.\Delta: \delta_n\.source = q_n \) do
      7. \( \text{Conditions} \leftarrow \text{Conditions} :: \delta_n\.\phi \);
      8. \( rs_d \leftarrow rs_d :: \delta_n\.\phi.rs \);

   /* ConstructMinTerms returns the min-terms from a set of conditions.
    * For example, if Conditions = \((\phi_1, \phi_2)\), then
    * MinTerms = \((\phi_1 \land \phi_2, \neg \phi_1 \land \phi_2, \phi_1 \land \neg \phi_2, \neg \phi_1 \land \neg \phi_2)\) */

   10. \( \text{MinTerms} \leftarrow \text{ConstructMinTerms}(\text{Conditions}) \);
5. foreach \( mt \in \text{MinTerms} \) do
   6. \( p_d \leftarrow \emptyset; W_d \leftarrow \emptyset \);
   7. foreach \( q_n \in q_d \) do
      8. foreach \( \delta_n \in A_n.\Delta: \delta_n\.source = q_n \) do
         9. /* \( \phi \models \psi \) denotes entailment, i.e., if \( \phi \) is true then \( \psi \) is necessarily also true. For example, \( \phi_1 \land \neg \phi_2 \models \phi_1 \). */
         10. if \( mt \models \delta_n.\phi \) then
            11. \( p_d \leftarrow p_d \cup \{\delta_n.target\} \);
            12. \( W_d \leftarrow W_d \cup \{\delta_n.W\} \);
5. \( \delta_d \leftarrow \text{CreateNewTransition}(q_d, mt(rs_d) \downarrow W_d \rightarrow p_d) \);
6. \( \Delta_d \leftarrow \Delta_d \cup \{\delta_d\} \);
7. \( q_{d,s} \leftarrow \{A_n.q_s\} \);
8. \( A_d \leftarrow (Q_d, q_{d,s}, Q_{f,d}, A_n.R, \Delta_d) \);
9. return \( A_d \);
We say that a register $r$ appears in a run at position $i$ if $r \in \delta_i W$.

**Proof:**

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

**Base case:** $k = 0$. Then $q_n = [1, q_{n,1}, \#] = [1, q_{n,s}, \#]$. The run $q_d = [1, q_{d,s}, \#]$ 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

\[
q_{n,k+1} = \cdots [k, q_{n,k}, v_{n,k}]
\]

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

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

\[
q_{d,k+1} = \cdots [k, q_{d,k}, v_{d,k}] \xrightarrow{\delta_{d,k}} [k + 1, q_{d,k+1}, v_{d,k+1}]
\]

By the induction hypothesis, we know that $q_{n,k} \in q_{d,k}$. By the construction Algorithm, we then know that, if $\phi_{n,k}^j = \phi_{n,k}^j'$ is the condition of a transition that takes the non-deterministic run to $q_{n,k+1}^j$, 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 $q_{n,k+1}^j$. More formally, $q_{d,k+1} = \bigcup_{j=1}^m q_{n,k+1}^j$.

As an example, see Figure 10. Figure 10a depicts part of a $nSRA$. Figure 10b depicts part of the $dSRA$ that would be constructed from that of Figure 10a. The construction algorithm would create the state $\{q_2, q_3\}$, the minterms from the conditions of all the outgoing transitions of $q_2$ and $q_3$ 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 10b. 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. (9) for the first of these runs and $q_{n,k} = q_4$ for the second. Assume also that both $\phi_1$ and $\phi_2$ are triggered after reading the $(k + 1)^{th}$ element, but not $\phi_3$. This means that the $nSRA$ would move to $q_2$ and $q_3$. In Eq. (9), this would mean that $m = 2$ and that $\delta_{n,k}^1\phi = \phi_1$ and $\delta_{n,k}^2\phi = \phi_2$. But in the $dSRA$ there is a transition that simulates this move of the $nSRA$. The minterm $\phi_1 \land \phi_2 \land \neg \phi_3$ moves the $dSRA$ to $\{q_2, q_3\}$. It contains $\delta_{n,k}^1\phi$ and $\delta_{n,k}^2\phi$ in their positive form and all other conditions (here only $\phi_3$) 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 \( \phi \) of the \( nSRA \) accesses are the same that this \( \phi \) accesses in the \( dSRA \) when participating in a minterm.

As we said, the condition on \( \delta_{d,k} \) is a conjunct (minterm), where all \( \phi_{n,k}^j \) 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:

\[
\phi = \phi_1(r_{1,1}, \ldots , r_{1,k}) \land \neg\phi_2(r_{2,1}, \ldots , r_{2,l}) \land \phi_3(r_{3,1}, \ldots , r_{3,m})
\]

This means that \( \phi_1(r_{1,1}, \ldots , r_{1,k}) \), with the exact same registers as arguments, will be the formula of a transition of the \( nSRA \) that was triggered. Similarly for \( \phi_3 \). With respect to \( \phi_2 \), 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 \( \delta_{d,k} \cdot \phi \) 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 \( \varrho_{n,k} \).

The second part of the proposition also holds, since, by the construction, \( \delta_{d,k} \) will write to all the registers that the various \( \delta_{n,k}^j \) 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 \( \delta_{d,k} \cdot W = \bigcup_{j=1}^{m} \delta_{n,k}^j \cdot W \) (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 9. 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 \( \delta_{n,k}^j \) 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 \( \varrho_n \) is equal to the last state of \( \varrho_d \). Thus, if \( \varrho_n \) reaches a final state, then the last state of \( \varrho_d \) will contain this final state and hence be itself a final state. Conversely, if \( \varrho_d \) reaches a final state of \( A_d \), it means that this state contains a final state of \( A_n \). Then, there must exist a \( \varrho_n \) that reached this final state.

\[ \square \]

### 10.11. Proof of Corollary 5.17

**Corollary 10.17.** Windowed \( SRA \) are closed under complement.

**Proof:**

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 \( q_{\text{dead}} \) (non-final) to \( A_d \). See Algorithm 9. For each state \( q \) of \( A_d \), we then gather all the conditions on its outgoing transitions. Let \( \Phi \) denote this set of conditions. We can then create the conjunction of all the negated conditions in \( \Phi \): \( \phi_{\text{dead}} := (\neg \phi_1) \land (\neg \phi_2) \land \cdots \land (\neg \phi_n) \), where \( \phi_i \in \Phi \) and \( \bigcup_{i=1}^{n} \phi_i = \Phi \). We then add a transition from \( q \) to \( q_{\text{dead}} \) with \( \phi_{\text{dead}} \) as its condition and \( \emptyset \) as its write registers. If we do this for every state \( q \in A_d \cdot Q \), we will have created a \( SRA \) that is equivalent to \( A_d \), since transitions to \( q_{\text{dead}} \) are only triggered if none of the other conditions in \( \Phi \) are triggered. If there exists a condition \( \phi_i \) that is triggered, the
Algorithm 9: Constructing the complement of a SRA (Complement).

**Input:** Windowed SRA $A$

**Output:** SRA $A_{\text{complement}}$ accepting the complement of $A$’s language

1. $A_d \leftarrow \text{Determinize}(A)$; // See Algorithm 8.
2. $q_{\text{dead}} \leftarrow \text{CreateNewState}();$
3. $\Delta_{\text{dead}} \leftarrow \emptyset;$
4. **foreach** $q \in A_d.Q$ **do**
   5. $\Phi \leftarrow \emptyset;$
   6. **foreach** $\delta \in A_d.\Delta : \delta.\text{source} = q$ **do**
      7. $\Phi \leftarrow \Phi \cup \delta.\phi$
   8. **end**
   9. $\phi_{\text{dead}} \leftarrow \top;$
   10. **foreach** $\phi_i \in \Phi$ **do**
    11. $\phi_{\text{dead}} \leftarrow \phi_{\text{dead}} \land (\neg \phi_i)$
    12. **end**
   13. $\delta_{\text{dead}} \leftarrow \text{CreateNewTransition}(q, \phi_{\text{dead}} \downarrow \emptyset \rightarrow q_{\text{dead}});$  
   14. $\Delta_{\text{dead}} \leftarrow \Delta_{\text{dead}} \cup \delta_{\text{dead}};$
   15. **end**
16. $\delta_{\text{loop}} \leftarrow \text{CreateNewTransition}(q_{\text{dead}}, \top \downarrow \emptyset \rightarrow q_{\text{dead}});$  
17. $\Delta_{\text{dead}} \leftarrow \Delta_{\text{dead}} \cup \delta_{\text{loop}};$
18. $Q_{\text{comp}} \leftarrow A.Q \cup \{q_{\text{dead}}\};$
19. $q_{\text{comp},s} \leftarrow A.q_s;$
20. $Q_{\text{comp},f} \leftarrow A.Q \setminus A.q_f;$
21. $R_{\text{comp}} \leftarrow A.R;$
22. $\Delta_{\text{comp}} \leftarrow A.\Delta \cup \Delta_{\text{dead}};$
23. $A_{\text{complement}} \leftarrow (Q_{\text{comp}}, q_{\text{comp},s}, Q_{\text{comp},f}, R_{\text{comp}}, \Delta_{\text{comp}});$  
24. **return** $A_{\text{complement}}$.
new automaton will behave exactly as $A_d$ and if no $ϕ$ is triggered it will go to $q_\text{dead}$. Now, if we add a self-loop transition on $q_\text{dead}$ with $⊤$ as its condition, we also ensure that the new automaton will always stay in $q_\text{dead}$ once it enters it. $q_\text{dead}$ 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 $A_{\text{complement}}$. This then ensures that if a string $S$ is accepted by $A$ (or $A_d$), it will not be accepted by $A_{\text{complement}}$ and if it is accepted by $A_{\text{complement}}$ it will not be accepted by $A$. This is indeed possible because $A$ (and $A_{\text{complement}}$) is deterministic and complete. Therefore, for every string $S$, there exists exactly one run of $A$ (and $A_{\text{complement}}$) over $S$. If $A$, after reading $S$, reaches a final state, $A_{\text{complement}}$ 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 $A_{\text{complement}}$, we would again reach $q_1$ and $q_2$, where, in this case, $q_1$ is final and $q_2$ is non-final. $A_{\text{complement}}$ would thus again accept $S$, which is not the desired behavior for $A_{\text{complement}}$. □

10.12. Proof of Proposition 6.2

**Proposition 10.18.** If $S = t_1, t_2, \cdots$ is a stream of elements from a universe $U$ of a $V$-structure $M$, where $t_i \in U$, and $e$ is a SREM over $M$, then, for every $S_{m..k}$, $S_{m..k} \in L(e)$ iff $S_{1..k} \in L(e_s)$ (and $S_{1..k} \in L(A_{e_s})$).

**Proof:**

First, assume that $S_{m..k} \in L(e)$ for some $m, 1 \leq m \leq k$ (we set $S_{1..0} = e$). Then, for $S_{1..k} = S_{1..(m-1)} \cdot S_{m..k}$, $S_{1..(m-1)} \in L(\top^*)$, since $\top^*$ accepts every string (sub-stream), including $e$. We know that $S_{m..k} \in L(e)$, thus $S_{1..k} \in L(\top^*) \cdot L(e) = L(\top^* \cdot e) = L(e_s)$. Conversely, assume that $S_{1..k} \in L(e_s)$. Therefore, $S_{1..k} \in L(\top^* \cdot e) = L(\top^*) \cdot L(e)$. As a result, $S_{1..k}$ may be split as $S_{1..k} = S_{1..(m-1)} \cdot S_{m..k}$ such that $S_{1..(m-1)} \in L(\top^*)$ and $S_{m..k} \in L(e)$. Note that $S_{1..(m-1)} = e$ is also possible, in which case the result still holds, since $e \in L(\top^*)$. □