Complex Event Forecasting

A Formal Framework

Elias Alevizos

NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Complex Event Forecasting: a Formal Framework

Elias Alevizos

ATHENS

January 2022

PhD THESIS

Complex Event Forecasting: a Formal Framework

Elias Alevizos

Examination Date: January 12, 2022

Abstract

As analytics moves towards a model of proactive computing, the requirement for forecasting acquires more importance. Systems with forecasting capabilities can play a significant role in assisting users to make smart decisions as soon as critical situations are detected. Being able to forecast that certain patterns in a stream have a high probability of being detected before they are actually detected can help an analyst focus early on what is important and possibly take a proactive action. The need for event forecasting as a means for proactive behavior has led to proposals about how forecasting could be conceptualized and integrated within a complex event processing system. However, such proposals still remain largely at a conceptual level, without providing concrete algorithms. On the other hand, there is a substantial body of work on the related fields of time-series forecasting, sequence prediction and temporal mining. However, time-series analysis is usually applied on numerical data streams, where each element of the stream corresponds to a measurement of some variable of interest. Moreover, these measurements are often assumed to take place at time intervals of constant length. Sequence prediction and temporal mining methods assume that their input is derived from finite sets of symbols. On the contrary, event processing systems need to be able to deal with symbolic/categorical streams, where each element might be accompanied by arguments, either numerical (thus the alphabet is possibly infinite) or symbolic, arriving at unspecified timepoints. The goal of this work is to provide a theoretical basis and build a prototype system for forecasting the occurrence of complex event patterns. This is be achieved by advancing the state-of-the-art in automaton models, going beyond classical automata, so that patterns can have the expressive power required by complex event processing applications. A probabilistic framework is used (fixed- or variable-order Markov models), so that the behavior of these automaton models may be quantified in a way that allows for producing forecasts with confidence. The resulting prototype is evaluated against synthetic datasets (for verification purposes) and against real-world datasets in order to demonstrate the applicability of the proposed system.

SUBJECT AREA: Formal Languages and Automata Theory, Databases, Artificial Intelligence

KEYWORDS: complex event processing, complex event recognition, symbolic automata, Markov chains, pattern Markov chains, finite automata, regular expressions, register automata, infinite alphabets, variable-order Markov models, prediction suffix trees

Acknowledgments

I would like to thank Panagiotis Rondogiannis, Professor at the National and Kapodistrian University of Athens, for his kind support and generosity while I was working on this thesis. I would also like to thank Georgios Paliouras, head of the SKEL lab at the National Center for Scientific Research (NCSR) "Demokritos". The work presented in this thesis was carried out under the auspices of NCSR "Demokritos" and the SKEL lab. It would not have been possible without their support. With his rock-solid presence and his wise oversight, George ensured that the work presented here as a complete thesis, would not get derailed (although the temptation at times was indeed quite strong). I also feel indebted to Alexander Artikis, Associate Professor at the University of Pireaus and the head of the Complex Event Recognition group at NCSR "Demokritos". His relentless attention to detail, his commitment to this project and his perseverance (even when it seemed that there was no point in persevering) were crucial for the completion of the work that makes up this thesis. I would also like to thank Minos Garofalakis (Professor at the Technical University of Crete), Antonios Deligiannakis (Associate Professor at the Technical University of Crete), Christos Doulkeridis (Associate Professor at the University of Pireaus) and Dimitrios Gunopoulos (Professor at the National and Kapodistrian University of Athens) for being kind enough to act as members of the seven-member examination committee. I was fortunate to collaborate with the first three on research projects and the output of this work (part of it) has found its way into this thesis. I would also like to express my gratitude to Panagiotis Stamatopoulos, Assistant Professor at the National and Kapodistrian University of Athens. Although I did not have the chance to work with him on any research related projects, I was lucky enough to collaborate with him as his teaching assistant. Takis is an exemplary teacher. Besides his technical competence (his classes have always been demanding), what distinguishes him as a teacher is a quality of his that is very rare indeed and that I value immensely: an acute sense of justice and the absence of any sense of privilege towards his students. I would be remiss not to mention Kostas Patroumpas and Nikos Giatrakos, both associate researchers at the "Athena" Research Center. They are exceptional collaborators and I would like to think that I played some role in our effort to produce some solid research results. Researchers have a tendency to construct "self-serving fetishizations" of the research activity itself by ignoring aspects of their work that do not seem to have a "direct" contribution in their research output. I would like to acknowledge here the importance of all the support that I received from the administrative/secretarial staff and colleagues at NCSR "Demokritos", especially from Dora Katsamori, Mariana Markouli, Sofoklis Karavellas, Nikos Katzouris and Evangelos Michelioudakis.

How can I ever thank I.M.M. and A.J.A.? It would be foolish on my part. How can you ever thank parts of your own self? From the very beginning of this voyage upon the Pequod, they were my radiant diamonds, my gigantic, cosmic gongs emitting the music of Aurora Borealis on dark heavens above tumultuous oceans. They taught me the most important lessons: that I must let go of the White Whale, that there is a hereafter toward which remembrance must go.

> *There are certain queer times and occasions in this strange mixed affair we call life when a man takes this whole universe for a vast practical joke, though the wit thereof he but dimly discerns, and more than suspects that the joke is at nobody's expense but his own. However, nothing dispirits, and nothing seems worth while disputing. He bolts down all events, all creeds, and beliefs, and persuasions, all hard things visible and invisible, never mind how knobby; as an ostrich of potent digestion gobbles down bullets and gun flints. And as for small difficulties and worryings, prospects of sudden disaster, peril of life and limb; all these, and death itself, seem to him only sly, good-natured hits, and jolly punches in the side bestowed by the unseen and unaccountable old joker. That odd sort of wayward mood I am speaking of, comes over a man only in*

some time of extreme tribulation; it comes in the very midst of his earnestness, so that what just before might have seemed to him a thing most momentous, now seems but a part of the general joke. There is nothing like the perils of whaling to breed this free and easy sort of genial, desperado philosophy; and with it I now regarded this whole voyage of the Pequod, and the great White Whale its object.

(H. Melville, Moby-Dick)

für welch unauffindbares Sein galt es da noch sich wach zu erhalten? für welche Zukunft galt da noch das unsägliche Bemühen um Erinnerung? in welche Zukunft sollte Erinnerung da noch eingehen? gab es da überhaupt noch Zukunft?
...
for what undiscoverable existence was still worth while to keep oneself awake? what future was worth this unspeakable effort to remember? what was the hereafter toward which remembrance must go? was there in reality any such hereafter?

(H. Broch, Der Tod des Vergil)

Contents

List of Figures

List of Tables

Part One

1. Introduction

Systems for Complex Event Pattern Matching, or Complex Event Recognition (CER), accept as input a stream of time-stamped, simple, derived events (SDE)s. A SDE (*low–level event*) is the result of applying a computational derivation process to some other event, such as an event coming from a sensor. Using SDEs as input, CER systems identify *complex events* (CE)s of interest– collections of events that satisfy some pattern. The definition of a CE (*high–level event*) imposes temporal and, possibly, atemporal constraints on its sub-events (SDEs or other CEs). Consider, for example, the recognition of attacks on computer network nodes, given the TCP/IP messages. A CER system attempting to detect a Denial of Service attack has to identify (as one possible scenario) both a forged IP address that fails to respond and that the rate of requests is unusually high. In maritime monitoring, in order to detect an instance of illegal fishing, a CER system has to perform both some geospatial tasks, such as estimating whether a vessel is moving inside a protected area, and temporal ones, like determining whether a vessel spent a significant amount of time in this area.

Numerous CER systems and languages have been proposed in the literature. These systems have a common goal, but differ in their architectures, data models, pattern languages, and processing mechanisms. For example, many CER systems provide users with a pattern language that is later compiled into some form of automaton. The automaton model is generally used to provide the semantics of the language and/or as an execution framework for pattern matching. Apart from automata, some CER systems employ tree-based models. Again, tree-based formalisms are used for both modeling and recognition, i.e., they may describe the complex event patterns to be recognized as well as the applied recognition algorithm. Recently, logic-based approaches to CER have been attracting considerable attention, since they exhibit a formal, declarative semantics, and at the same time have been proven efficient enough for Big Data applications. The avalanche of streaming data in the last decade has sparked an interest in CER technologies capable of processing high-velocity data streams by using one of the above mentioned formalisms [42, 64]. Due to the nature of the task of CER, Complex Events must be detected with minimal latency.

As a result, a significant body of work has been devoted to computational optimization issues.

Less attention has been paid to forecasting event patterns [64], despite the fact that forecasting has attracted considerable attention in various related research areas, such as time-series forecasting [99], sequence prediction [24, 28, 37, 121, 140], temporal mining [34, 83, 135, 144] and process mining [97]. As analytics moves towards a model of proactive computing, the requirement for Complex Event Forecasting (CEF) acquires more importance [55]. Systems with forecasting capabilities can play a significant role in assisting users to make smart decisions as soon as critical situations are detected.

Consider, for example, the domain of credit card fraud management [21], where the detection of suspicious activity patterns of credit cards must occur with minimal latency that is in the order of a few milliseconds. The decision margin is extremely narrow. Being able to forecast that a certain sequence of transactions is very likely to be a fraudulent pattern provides wider margins both for decision and for action. For example, a processing system might decide to devote more resources and higher priority to those suspicious patterns to ensure that the latency requirement will be satisfied. The field of moving object monitoring (for ships at sea, aircrafts in the air or vehicles on the ground) provides yet another example where CEF could be a crucial functionality [137]. Collision avoidance is obviously of paramount importance for this domain. A monitoring system with the ability to infer that two (or more) moving objects are on a collision course and forecast that they will indeed collide if no action is taken would provide significant help to the relevant authorities. CEF could play an important role even in in-silico biology, where computationally demanding simulations of biological systems are often executed to determine the properties of these systems and their response to treatments [109]. These simulations are typically run on supercomputers and are evaluated afterwards to determine which of them seem promising enough from a therapeutic point of view. A system that could monitor these simulations as they run, forecast which of them will turn out to be non-pertinent and decide to terminate them at an early stage, could thus save valuable computational resources and significantly speed-up the execution of such in-silico experiments. Note that these are domains with different characteristics. For example, some of them have a strong geospatial component (monitoring of moving entities), whereas in others this component is minimal (in-silico biology). Domain-specific solutions (e.g., trajectory prediction for moving objects) cannot thus be universally applied. We need a more general framework.

The need for CEF as a means for proactive behavior has led to proposals about how forecasting could be conceptualized and integrated within a complex event processing system. However, such proposals still remain largely at a conceptual level, without providing concrete algorithms [20, 36, 53, 60]. In this thesis, we present a formal framework for CEF, along with an implementation and extensive experimental results on real and synthetic data from diverse application domains. Our framework allows a user to define a pattern for a complex event, e.g., a pattern for fraudulent credit card transactions or for two moving objects moving in close proximity and towards each other. 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.

This thesis advances the state-of-the-art along the following dimensions:

- It proposes a CEF framework that is both formal and easy to use. CER frameworks often lack clear semantics, which in turn leads to confusion about how patterns should be written and which operators are allowed. This problem is exacerbated in CEF, where a formalism for defining the patterns to be forecast may be lacking

completely. Our framework is formal, compositional and as easy to use as writing regular expressions. The only basic requirement is that the user declaratively define a pattern and provide a training dataset.
- It proposes a language for CER, along with a computational model for patterns written in this language, 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. This is achieved by using automata with memory. Although similar languages for CER have been proposed in the past, to the best of our knowledge, this is the first time that a full (with memory), formal, compositional automata-based framework for CER is proposed which has clear semantics and whose computational model may be systematically investigated.
- It proposes the combination of automata with Markov models so as to provide a probabilistic description for the behavior of automata and infer in an online manner if and when a Complex Event is expected to occur.
- It shows how our proposed framework can move beyond fixed-order Markov models in order to uncover deep probabilistic dependencies in a stream by using variable-order Markov models. By being able to look deeper into the past, we achieve higher accuracy scores compared to other state-of-the-art solutions for CEF.
- Our framework can perform various types of forecasting and thus subsumes previous methods that restrict themselves to one type of forecasting. It can perform both simple event forecasting (i.e., predicting what the next input event might be) and Complex Event Forecasting (events defined through a pattern).

The general outline of the thesis is depicted in Figure 1.1. The thesis revolves around two main axes: one focusing on the expressivity of the (already existing or proposed herein) CER languages and the other on the probabilistic framework used in order to make inferences about the behavior of the employed automata. The thesis first begins with Chapter 2, where we present the necessary background for CER concerning the various models and languages that have been proposed in the literature. In Chapter 3 we provide an overview of the previously proposed methods for CEF. We then move to the main parts of the thesis. Chapter 4 presents our first attempt towards CEF. It uses classical automata and fixed-order Markov models (k-order Markov chains). This chapter corresponds to the first (out of a total of four), starting from the top, dark connection in Figure 1.1, the one that connects the orange node labeled *"Classical finite automata"* with the blue node labeled *"Fixed-order Markov models"*. We extend this framework in Chapter 5, where we show how we can improve the expressive power of our system by using symbolic (instead of classical) automata. This chapter corresponds to the second connection in Figure 1.1. Chapters 6 and 7 improve our framework by describing how variable-order Markov models may be combined with symbolic automata in order to perform efficient CEF while retaining high accuracy scores. The former presents the theoretical background of this method whereas the latter presents the relevant experimental results. These chapters corresponds to the third connection in Figure 1.1. Finally, Chapters 8, 9 and 10 describe yet another theoretical extension. We show how symbolic automata can be enriched with registers in order to allow for patterns where constraints among multiple events need to be imposed. We show that our proposed model does not retain all the nice closure properties of symbolic (and classical) automata, but still remains a powerful model for the purposes of CER/F. These chapters correspond to the final, fourth connection in Figure 1.1.

This thesis is based on the following publications:

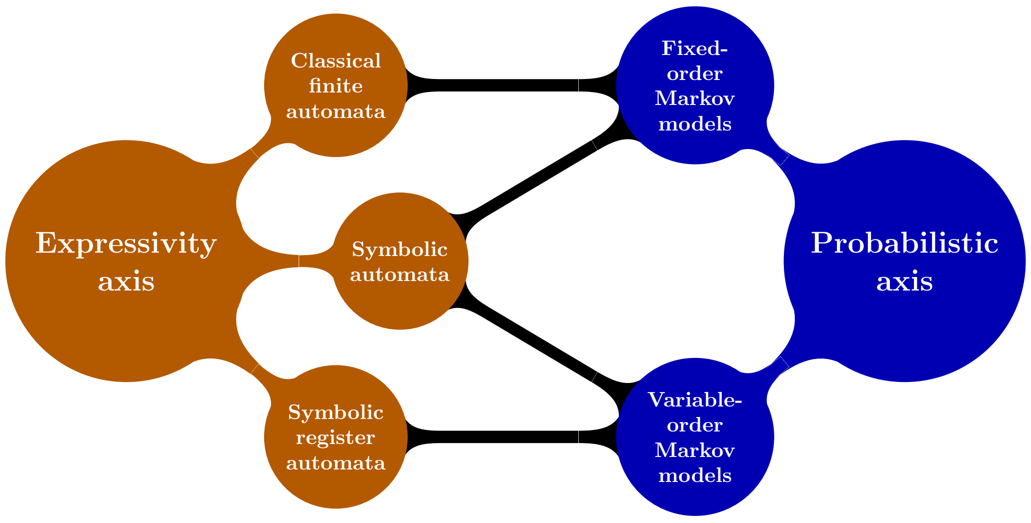

Classical finite automata
Fixed-order Markov models
Expressivity axis
Symbolic automata
Probabilistic axis
Symbolic register automata
Variable-order Markov models

Figure 1.1: General architecture and overview of the thesis.

1. Main
   (a) Alevizos, Artikis, Paliouras, **Symbolic Register Automata for Complex Event Recognition and Forecasting**, submitted as journal article
   (b) Alevizos, Artikis, Paliouras, **Complex Event Forecasting with Prediction Suffix Trees**, *VLDB journal*, 2022 [10]
   (c) Alevizos, Artikis, Paliouras, **Wayeb: a Tool for Complex Event Forecasting**, *LPAR*, 2018 [8]
   (d) Alevizos, Artikis, Paliouras, **Event Forecasting with Pattern Markov Chains**, *DEBS*, 2017 [6]
2. Surveys
   (a) Giatrakos, Alevizos, Artikis, Deligianakis, Garofalakis, **Complex Event Recognition in the Big Data Era: a Survey**, *VLDB journal*, 2020 [64]
   (b) Alevizos, Skarlatidis, Artikis, Paliouras, **Probabilistic Complex Event Recognition: a Survey**, *ACM Computing Surveys*, 2017 [11]
3. Peripheral (demos and application papers)
   (a) Ntoulias, Alevizos, Artikis, Akasiadis, Koumparos, **Online Fleet Monitoring with Scalable Event Recognition and Forecasting**, *GeoInformatica*, 2022 [106]
   (b) Vodas, Bereta, Kladis, Zissis, Alevizos, Ntoulias, Artikis, Deligiannakis, Kontaxakis, Giatrakos, Arnu, Yaqub, Temme, Torok, Klinkenberg, **Online Distributed Maritime Event Detection and Forecasting over Big Vessel Tracking Data**, *IEEE Big Data*, 2021 [136]
   (c) Ntoulias, Alevizos, Artikis, Koumparos, **Online Trajectory Analysis with Scalable Event Recognition**, *EDBT Workshops*, 2021 [105]
   (d) Vouros, Vlachou, Santipantakis, Doulkeridis, Pelekis, Georgiou, Theodoridis, Patroumpas, Alevizos, Artikis, Fuchs, Mock, Andrienko, Andrienko, Claramunt, Ray, Camossi, Jousselme, **Increasing Maritime Situation Awareness via Trajectory Detection, Enrichment and Recognition of Events**, *W2GIS*, 2018 [138]
   (e) Qadah, Mock, Alevizos, Fuchs, **A Distributed Online Learning Approach**

for Pattern Prediction over Movement Event Streams with Apache Flink, *EDBT Workshops*, 2018 [117]

(f) Vouros, Vlachou, Santipantakis, Doulkeridis, Pelekis, Georgiou, Theodoridis, Patroumpas, Alevizos, Artikis, Fuchs, Mock, Andrienko, Andrienko, Claramunt, Ray, Camossi, Jousselme, Scarlatti, Cordero, **Big Data Analytics for Time Critical Mobility Forecasting: Recent Progress and Research Challenges**, *EDBT*, 2018 [137]

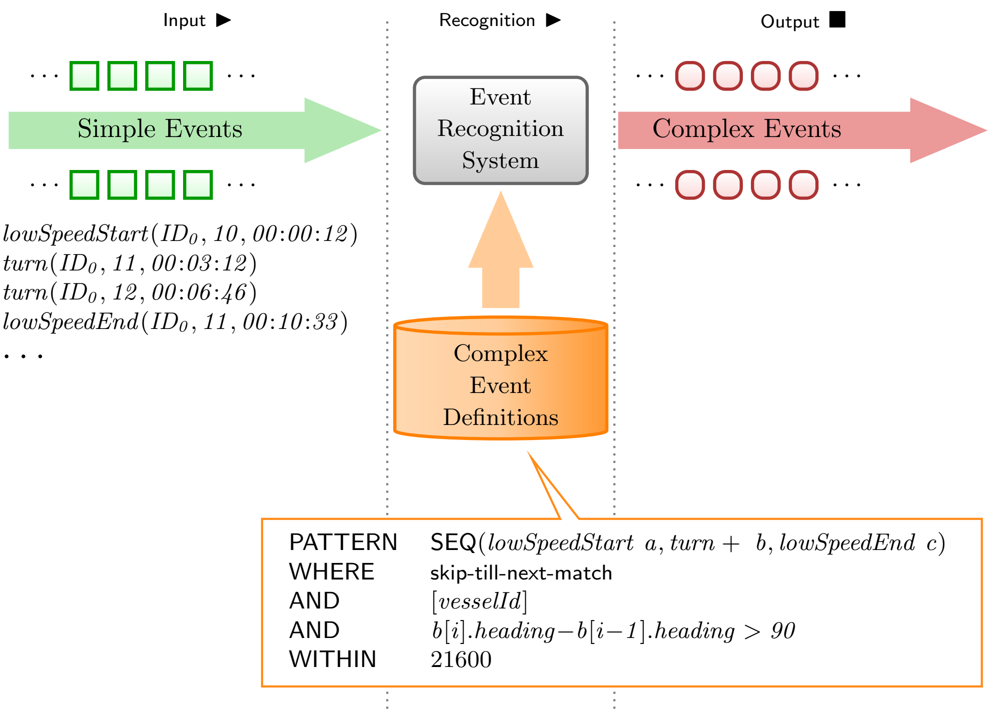
Chapters 4 - 10 have been based on the material of the first set of publications. Publication 2a has provided material for Chapters 2 and 3. Publication 2b provides an overview of CER systems that can work under uncertainty. Even though its material is relevant to this thesis, we decided not to include it here in order to minimize incidents of "attention overflow" on the part of the readers. The included material is more than enough for the purposes of the thesis. Note, also, that no material from the third set of publications has been included. These publications present how our CEF system, Wayeb, has been extended towards various directions (e.g., distributed versions) or has been incorporated in complex architectures. They thus do no constitute integral parts of the work presented herein. Their inclusion would render our thesis obese.

2. Background

Our goal in this chapter is to provide an overview of the various languages that have been proposed, determine the regions of their convergence and divergence, and establish requirements that they should satisfy in terms of their expressive power (for a short tutorial on CER languages see also [22]). It is not our intention to present a survey of CER systems, since this is covered elsewhere [42]. Our focus is on identifying the classes of languages typically used in CER in order to determine their expressive power and limits, and to initiate a discussion about how expressive power may interact with the performance exhibited by a CER system. We begin by presenting a set of language features and notions usually encountered in CER systems, as well as a set of extra requirements that we deem should be satisfied by such systems but have attracted less attention thus far. We then present the three classes of CER systems that we have identified: automata-based systems, logic-based ones and those that employ trees. Note that there also exist hybrid systems, albeit these constitute a minority. We briefly mention those as well.

2.1 Setting the scene

Due to the great variety of existing CER languages and systems, there is a lack of a common ground for comparing them, and extracting a set of common operators is far from being a trivial task. Notwithstanding this disparity, it seems that it is indeed possible to identify some basic features that should be present in every CER language; in fact, as of late there have appeared attempts targeting such a unification and homogenization [67, 73]. Therefore, before delving into the presentation of the languages themselves, we begin by presenting these common features and establishing a set of requirements. In Section 2.1.1 we discuss the basic operators that constitute the building blocks of a CER language, borrowing from [11]. In Sections 2.1.2 – 2.1.5 we discuss some extra features that are common in CER systems and conclude with a discussion on a set of functionality requirements that are less frequently satisfied.



Input
Recognition
Output
Simple Events
Event Recognition System
Complex Events
Complex Event Definitions
$lowSpeedStart(ID_0, 10, 00{:}00{:}12)$
$turn(ID_0, 11, 00{:}03{:}12)$
$turn(ID_0, 12, 00{:}06{:}46)$
$lowSpeedEnd(ID_0, 11, 00{:}10{:}33)$

```
PATTERN   SEQ(lowSpeedStart a, turn + b, lowSpeedEnd c)
WHERE     skip-till-next-match
AND       [vesselId]
AND       b[i].heading − b[i−1].heading > 90
WITHIN    21600
```

Figure 2.1: High-level view of a CER system, using the maritime domain as an example. The simple event stream consists of *turn*, *lowSpeedStart* and *lowSpeedEnd* events from a vessel with id ID_0. The pattern attempts to capture a sequence of events, where the first indicates that a vessel starts moving at a low speed, then the vessel performs one or more turns and finally ends by a single event indicating the end of the slow movement.

2.1.1 Abstract event algebra

A CER system takes as input a stream of events, also called simple derived events (SDEs), along with a set of patterns, defining relations among the SDEs, and detects instances of pattern satisfaction, thus producing an output stream of complex events [56, 93]. Since time is of critical importance for CER, 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 CE.

Typically, an event has the structure of a tuple of values which might be numerical or categorical. The most important attributes which are always to be found in an event are those of *Event Type* and *timestamp*. The timestamp may be a single timepoint, indicating the occurrence time of the event, or an interval, in cases where events may be durative. These two basic attributes may be accompanied by any number of extra attributes. As an example, consider the domain of maritime monitoring where the input stream consists of events emitted from vessels sailing at sea and relaying information about their kinematic behavior, e.g., location, speed, heading, etc [113]. In the terminology of the maritime domain, these are called AIS (Automatic Identification System) messages. Each such SDE may contain an event type referring to the type of movement executed by a vessel (e.g., turning, sailing, accelerating) and a timestamp. Additionally, it may contain a number uniquely identifying each vessel (an identifier) along with attributes for its longitude, latitude, speed, etc. Figure 2.1 depicts a high-level view of a CER system, using the maritime domain as an example.

In order to define the CEs to be detected upon the stream of SDEs, we need a language that is expressive enough for the needs of CER. The most basic operator is that of *selection* according to a set of predicates. This set of selection predicates are applied to every SDE and those SDEs that do not satisfy the predicates are filtered out. An example of a selection operator on the AIS messages could be one that checks the speed of each vessel and retains only those messages with a speed above 0.1 knots in order to keep only those vessels that are actually on the move. As far as the temporal operators are concerned, the most basic is the *sequence* operator, usually denoted by a semicolon. The implied constraint in this case is that the events connected through a sequence operator must succeed one another temporally. These operators are sufficient to define simple patterns, but for more complex patterns, we need to incorporate some more operators. With the help of the theory of descriptive complexity, recent work has identified those constructs of an event algebra which strike a balance between expressive power and complexity [143]. For other event formalisms, see also [12, 13, 29, 61, 67, 82].

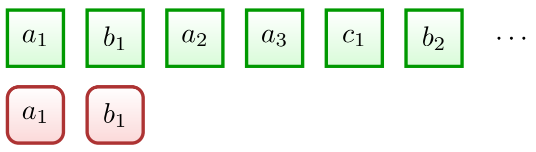
These constructs may be summarized as follows:

- *Sequence* Two events following each other in time.
- *Disjunction*: Either of two events occurring, regardless of their temporal relation.
- *Iteration*: An event occurring N times in sequence, where $N \geq 0$.
- *Conjunction*: Both events occurring, regardless of their temporal relation.
- *Negation*: Absence of event occurrence.
- *Selection*: Select those events whose attributes satisfy a set of predicates/relations, temporal or otherwise.
- *Projection*: Return an event whose attribute values are a possibly transformed subset of the attribute values of its sub-events.
- *Windowing*: The event pattern must occur within a specified time window.

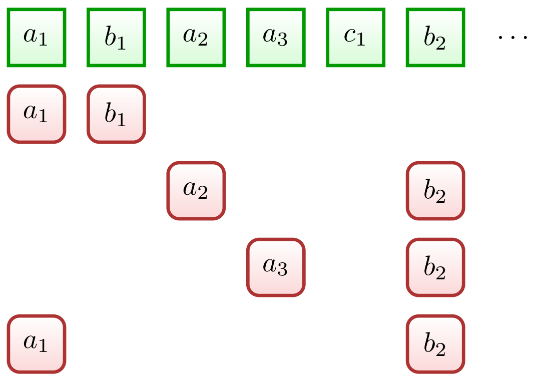
The above list can be presented in the form of a simple event algebra, as presented below [11]

$$
\begin{aligned}
ce ::= \;& sde && \mid \text{Base case}\\
& ce_1 \; ; \; ce_2 && \mid \text{Sequence}\\
& ce_1 \; \vee \; ce_2 && \mid \text{Disjunction}\\
& ce^* && \mid \text{Iteration}\\
& ce_1 \; \wedge \; ce_2 && \mid \text{Conjunction}\\
& \neg \; ce && \mid \text{Negation}\\
& \sigma_\theta(ce) && \mid \text{Selection}\\
& \pi_m(ce) && \mid \text{Projection}\\
& [ce]_{T_1}^{T_2} && \mid \text{Windowing (from } T_1 \text{ to } T_2)
\end{aligned}
\tag{2.1}
$$

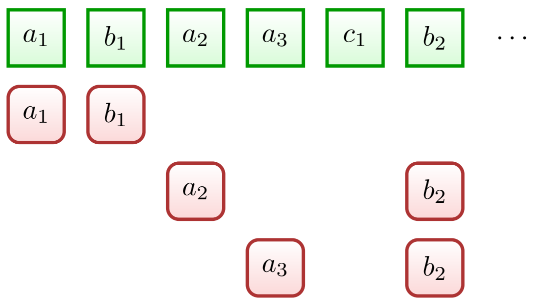
where $\sigma_\theta(ce(v_1,\ldots,v_n))$ *selects* those ce whose variables v_i satisfy the set of predicates θ and $\pi_m(ce(a_1,\ldots,a_n))$ *returns* a ce whose attribute values are a possibly transformed subset of the attribute values of a_i of the initial ce, according to a set of mapping expressions m. Note that *conjunction* may also be written by combining sequence and disjunction, as: $ce ::= (ce_1;ce_2) \vee (ce_2;ce_1)$. Please note that, compared to the event algebra presented in [11], we have explicitly added the conjunction operator, as it is not only important, but it may also require special handling and cannot always be derived from the other operators when there is no support for disjunction. This inductive definition showcases an important feature of CER languages: their compositionality, i.e., the ability to define hierarchies of events, where SDEs may be used to define some CEs and these may again be used to



(a) Matches under strict-contiguity.



(b) Matches under skip-till-any-match.



(c) Matches under skip-till-next-match.

Figure 2.2: Example of selecting input events for the pattern $R = a;b$ under different selection policies. The top stream (green rectangles) represents the input stream. The bottom streams (red, rounded rectangles) represent the matches produced, one per row.

define other, higher-level CEs.

2.1.2 Selection policies

The first three temporal operators in the event algebra presented above (Eq. (2.1)), namely sequence, disjunction and iteration, resemble the three operators of standard regular expressions: concatenation, union and Kleene star respectively. This similarity is not a coincidence, as automata have frequently been used as computational models in CER. Since SDEs are not symbols but tuples, the automata variations employed in CER typically have predicates on their transitions: the predicates of the selection operators. By following the semantics of regular expressions, one would then expect that the SDEs involved in a match of a CER pattern should occur contiguously in the input stream. As an example, consider the case where we have a stream with event types a, b or c (for simplicity we ignore all other attributes) and we define a simple pattern as $a;b$ (an event of type a followed by one of type b). Figure 2.2a shows an example of such a stream along with the match that would be detected.

However, in CER it is often the case that we are also interested in matches where the involved SDEs need not be contiguous. This is where the notion of *selection policies* enters the scene [42, 143, 145]. As its name suggests, a selection policy determines which SDEs may be allowed to enter a match by establishing conditions about whether we are allowed to skip any events, deemed as "irrelevant", or not. The single match of Figure 2.2a is the result of our pattern under one such policy, called strict-contiguity, due to its requirement that all events in a match must be contiguous in the input stream. This is indeed the strictest policy in the sense that it produces the fewest matches and is the one most closely associated to standard regular expressions. On the other end of the strictness spectrum is the so called skip-till-any-match policy. In this case, any combination of events that satisfy the succession constraints of the pattern, regardless of whether they are contiguous or not, is considered a match. This policy is closer in spirit to logic programming where running a query/goal returns all results that satisfy it. Figure 2.2b depicts the matches of our example stream for the pattern $a;b$ under the skip-till-any-match policy. All combinations of a events followed by b events are produced as matches.

In between these two extremes, there exists the skip-till-next-match policy. In this case, it is still possible to skip irrelevant events, e.g., a c event occurring between an a and a b, but only the immediately next relevant event in the stream is selected, thus restricting the number of matches with respect to the skip-till-any-match policy. Figure 2.2c shows the matches produced in our example under skip-till-next-match. The match $\{a_1,b_2\}$ of skip-till-any-match is no longer a match, since the partial match that started with a_1 selected b_1 and then capitulated.

Another common policy is the so-called partition-contiguity policy, where the stream is first partitioned into substreams according to a predicate and then strict-contiguity is applied to each substream separately. For example, for the maritime domain this could be useful in order to partition the stream according to the vessel identifier so that each vessel has its own stream and a pattern may be applied to each individual vessel. The same partitioning technique can also be applied to any of the previous three selection policies. In the field of runtime verification a similar technique is used, under the name of "parametric trace slicing" [33]. Within the field of CER itself, such partitioning schemes may be subsumed under the notion of *contexts* [56, 145]. A context may be defined as a specification of conditions that groups events together for purposes of common processing where each event is assigned to one or more context partitions [56]. As a result, partition-contiguity may not necessarily be viewed as a separate selection policy, but as a combination of a selection policy (strict-contiguity) with a partition/slicing/context scheme.

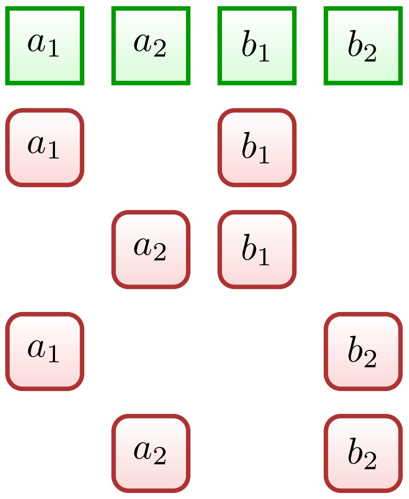
As a closing remark to this section, we should note that there is no universal consensus about the semantics of selection policies. The discussion of this section borrows the terminology and semantics of the SASE CER engine [3, 143]. FlinkCEP [15], a CER engine built on top of the Flink distributed processing engine [14], uses very similar notions for selecting events, but with different semantics in some cases. We will discuss this issue in more detail in the following sections.

2.1.3 Consumption policies

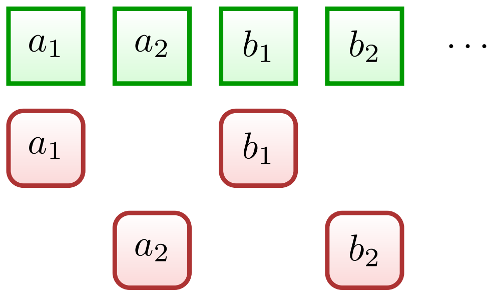
There is yet another notion for determining which events may participate in a match: that of *consumption policies* [42, 56, 75, 145]. A consumption policy determines whether an event that has participated in match of a pattern R is allowed to participate again in other matches of R. In what follows, we adopt the terminology of [56].

The most relaxed consumption policy is called the reuse policy. As its name suggests, under this policy events may be used without any restrictions to any number of matches, provided that they satisfy the constraints of the pattern and of the selection policy. Figure 2.3a shows an example of the matches produced for the pattern $a;b$ under the reuse consumption policy and the skip-till-any-match selection policy.

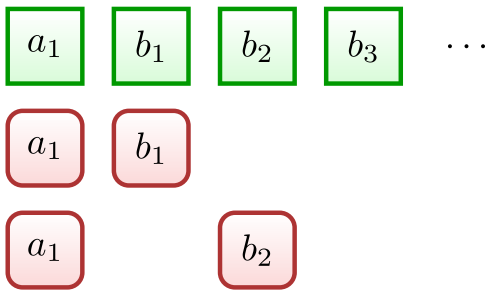
The strictest consumption policy is called consume. Under consume, whenever an event becomes member of a match it is no longer allowed to be included in any future matches. Figure 2.3b shows an example of this policy. Upon the arrival of b_1, the candidate



(a) Matches under reuse.



(b) Matches under consume.



(c) Matches under bounded-reuse.

Figure 2.3: Example of consuming input events for $R = a;b$ under different consumption policies, with skip-till-any-match as the selection policy. The top stream (green rectangles) represents the input stream. The bottom streams (red, rounded rectangles) represent the matches produced, one per row. Note that for Figure 2.3c the input stream is slightly different.

matches are $\{a_1, b_1\}$ and $\{a_2, b_1\}$. Assuming that the production of the matches starts from the one whose initiator (a) is temporally first (for other options, see [145]), then $\{a_1, b_1\}$ is produced. This has two effects: a) b_1 becomes no longer available and $\{a_2, b_1\}$ is disqualified as a match; b) a_1 also becomes invalidated, thus disqualifying $\{a_1, b_2\}$ as a match when b_2 arrives at the next timepoint. We are therefore left with two matches after b_2: $\{a_1, b_1\}$ and $\{a_2, b_2\}$.

An intermediate policy is the one called the bounded-reuse consumption policy. The goal of this policy is to allow the reuse of events, but to impose an upper bound on the number of matches in which an event may participate. Figure 2.3c shows the matches produced under this policy when at most 2 matches are allowed. Assuming again the production of matches follows a temporal order, the last match that would normally be produced under reuse, $\{a_1, b_3\}$, is dropped since 2 matches with a_1 have already been produced.

2.1.4 Windows

CER systems are not expected to detect CEs by considering at every timepoint all SDEs that have occurred in the past. In order to limit their search space, which can quickly become unmanageable, especially when relaxed selection policies and consumption policies are used, they typically incorporate a special operator, that of *windowing* [42, 56]. Windows are usually applied on a per pattern basis and their function is to restrict, up to a certain point in the past, the SDEs to be considered. Although they can in principle be subsumed as a constraint built from the standard operators of an event algebra (by restricting the time difference between the last and first events in a match), they are usually defined as an extra operator, due to their importance and their effect on the complexity of pattern evaluation.

The most typical window constraint to be found in a pattern is of the form $within(W)$, usually appended at the end. A key distinction between window types is the one between time-based and count-based (also called tuple-based) windows. Time-based windows impose a constraint on the size of the time interval into which a match can extend. The constraint imposed is that a (temporally ordered) candidate match $M = \{e_1, \ldots, e_n\}$ is indeed a valid match if $e_n.timestamp - e_1.timestamp < W$. This basically constitutes a sliding window of length W whose step (slide) is equal to the temporal resolution of the CER system. On the other hand, tuple-based windows impose an explicit constraint. In this case, a constraint like $within(W)$ cannot be directly expressed through the operators of an event algebra; it implies that only the last W SDEs that have arrived at the system are to be considered, regardless of their timestamps. Some CER systems also include other window types, like tumbling windows [30].

From an implementation point of view, we may also distinguish between actual and logical windowing mechanisms. With actual windows, events belonging to a window are buffered and their processing begins as soon as the window's timer has expired, for time-based windows, or the count limit has been reached for count-based ones. When logical windowing is used, the SDEs are not buffered, but are processed as soon as they arrive.

2.1.5 Requirements

Our discussion thus far has focused on a basic core of operators and features for a CER language. We conclude this section by adducing a set of extra requirements for CER, extracted from the limitations of the core features.

Support for both instantaneous and durative events: The majority of CER languages make the assumption that the timestamp of each event, either SDE or CE, is a single timepoint. However, there are domains where it is more natural to express events as having a temporal duration. For example, in human activity recognition, the activity of a person walking is durative. The same holds in the maritime domain for several types of vessel behavior, such as fishing. Sometimes, the interval of an activity may be open, in the sense that it is still ongoing. For example, we should not wait until the end of fishing before we report it. Additionally, there are some subtle issues with respect to the semantics of instantaneous events. When single timepoints are used as timestamps, it is possible that some unintended semantics might be introduced [61, 111] (see also Sections 2.4 and 2.6). Note also that formalisms for reasoning on durative events have appeared in the past, such as the Event Calculus [29, 82] and Allen's Interval Algebra [12, 13], and have been used for defining event algebras (e.g., [18, 112]).

Support for relational events: By relational events we mean CEs whose detection depends on multiple entities of the domain under study. In the maritime domain, detecting a possible collision requires to relate the activity of at least two vessels. It is possible to detect such relational CEs by partitioning the input stream according to its entities and attempt to join these substreams. Such joins raise new issues with respect to the runtime complexity of CE patterns. For instance, relational events are not easy to be expressed and captured with simple computational models, like simple automata or even extensions of automata often used in CER, which often assume that only a single stream exists. Therefore, more expressive models are required, like quantified event automata, used for runtime verification [23].

Support for concurrency constraints: Sequence is one of the basic operators in CER and this is the reason why automata are so popular as computational models for CER. On the other hand, there exist patterns that require a mechanism for detecting concurrent events, especially in the case of relational CEs. The above mentioned example of collision detection is such a case, since a pattern for it would need to relate the behavior of two vessels at the same time.

Support for patterns without windows: As already mentioned, windows are essential in CER since they significantly reduce the search space for pattern matching. Although they might also be useful from a conceptual point of view as well (there are indeed patterns which are meaningless if they extend beyond a time interval), it is also the case that some long-term relationships are important to capture as a CE, without knowing beforehand their maximum temtypeporal duration. For example, a pattern for detecting when a vessel approaches a port cannot be meaningfully constrained in terms of its duration, since different vessel types exhibit different movement patterns (or the window would have to be so large in order to include all cases, rendering it essentially meaningless). Although it is conceptually and semantically possible to get rid of the window constraint, this would incur a heavy performance cost, considering that the size of the window is one of the main factors affecting runtime complexity (the larger the window the more partial matches that need to be maintained) [143].

Support for event hierarchies: We have already mentioned that it is important to be able to define CEs hierarchically, i.e., using lower-level CEs to define other CEs at higher levels. Hierarchies allow for structured, succinct representations, and thus code maintenance. We repeat this requirement here, since it is not always satisfied. Event hierarchies raise issues both with respect to the semantics of CER languages and the performance of CER systems. For example, CER systems that are based on automata resemble in certain respects register automata [79], i.e., automata that are equipped with registers in order to store past elements of a stream and later be able to retrieve them for comparison purposes. However, register automata are not closed under complement, which implies that it is not obvious how the negation operator is to be properly used in a CER language that uses automata as its underlying model. With respect to performance, a hierarchy of CEs might exhibit a structure where CEs might participate in the definition of multiple other CEs. Treating such hierarchies in a naive manner would result in redundant computations. For hierarchies to be a viable feature, careful optimizations should be employed [18, 90].

Support for background knowledge and non-temporal reasoning: Most CER systems focus on temporal reasoning and their selection predicates are usually (in)equalities on event attributes or aggregate functions, like *averaging* of a certain event attribute, when iteration is present. Useful as this kind of reasoning might be, there are also cases where we need to take into account information that is not present in the SDEs themselves. For example, we might need to know whether it is prohibited to fish within a specific area at sea in order to detect vessels that violate this restriction. Therefore, it is important for a CER system to be able to incorporate background knowledge (e.g., areas where fishing is prohibited) and to perform non-temporal reasoning as well (e.g., that a position lies within a given area polygon).

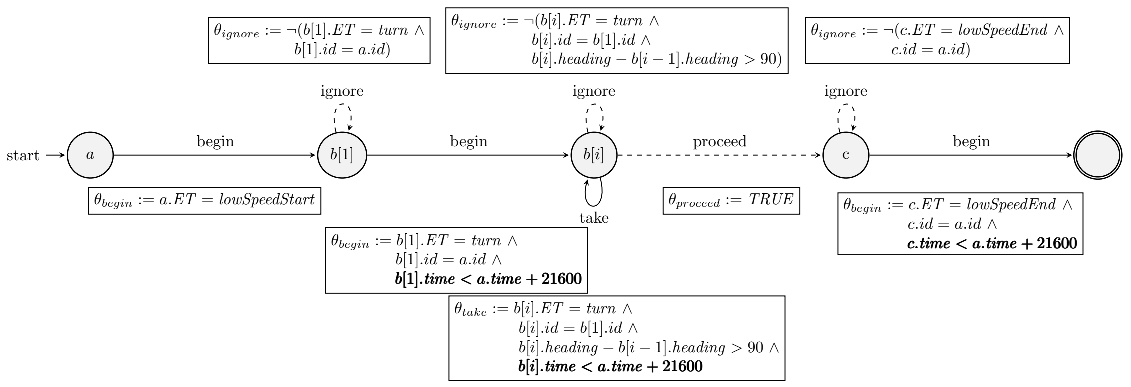

$\theta_{ignore} := \neg(b[1].ET = turn \wedge b[1].id = a.id)$
$\theta_{ignore} := \neg(b[i].ET = turn \wedge b[i].id = b[1].id \wedge b[i].heading - b[i-1].heading > 90)$
$\theta_{ignore} := \neg(c.ET = lowSpeedEnd \wedge c.id = a.id)$
ignore
ignore
ignore
start
a
begin
$b[1]$
begin
$b[i]$
proceed
c
begin
take
$\theta_{begin} := a.ET = lowSpeedStart$
$\theta_{proceed} := TRUE$
$\theta_{begin} := c.ET = lowSpeedEnd \wedge c.id = a.id \wedge \mathbf{c.time < a.time + 21600}$
$\theta_{begin} := b[1].ET = turn \wedge b[1].id = a.id \wedge \mathbf{b[1].time < a.time + 21600}$
$\theta_{take} := b[i].ET = turn \wedge b[i].id = b[1].id \wedge b[i].heading - b[i-1].heading > 90 \wedge \mathbf{b[i].time < a.time + 21600}$

Figure 2.4: A non-deterministic automaton (NFA), as constructed by SASE, for the pattern of Figure 2.1. *begin* edges move the NFA to a next state and store the SDE. *take* edges iterate over the same state, again storing the SDEs. *ignore* edges skip "irrelevant" events due to the use of skip-till-next-match. The *proceed* takes the NFA out of the iteration. Above or below each edge, its respective constraints are shown as a (negated) conjunction of predicates found in the pattern. Conjuncts shown in bold correspond to the window constraint. Note that the window constraint is placed on multiple edges in order to be able to "kill" an instance of the NFA as soon as possible if the constraint is violated, without waiting until it reaches the last non-final state (c). *ET* stands for *EventType*.

2.2 Automata-based systems

Since the temporal operators of CER languages (sequence, disjunction, iteration) resemble those of regular expressions, it is no surprise that automata have been quite popular as computational models of CER systems. In such systems, patterns are usually defined in a language similar to SQL, with the addition of operators for the regular part of the pattern. This regular part typically appears first, followed by a set of predicates that the events appearing in the regular part must satisfy. After a pattern has been defined, it goes through a compiler that transforms it to an automaton (often non-deterministic). The automaton is then fed with the stream of SDEs, changing states according to whether predicates on the outgoing transitions of its current state are satisfied. Upon the triggering of a transition, the responsible SDE may be ignored if it is irrelevant or stored if it is relevant. Whenever it reaches a final state, we say that a full match has been completed and a CE is detected. The user is then informed of the occurrence of the CE, along with the SDEs participating in the match. When the automaton is in a non-final state, we say that its stored SDEs constitute a "partial match" that may or may not eventually lead to a full match.

Automata-based CER systems started as academic projects (e.g., Cayuga [45, 46], SASE [3, 72, 141, 143] and its derivatives, like SASE+ [47], NextCEP [124], DistCED [115]), inspired to some extent by previous work on Data Stream Management Systems, such as TelegraphCQ [31], CQL [17] and CQL's commercial derivative, Oracle CQL [108]. Work along these lines still continues to this date [8], with effort being also devoted to providing solid foundations [67]. Automata-based CER has recently reached a new level of maturity and found its way into systems, such as Flink, which provides a library for complex event recognition and processing, called FlinkCEP [15]. In what follows, we will focus mainly on the popular FlinkCEP and the highly cited SASE, which are both open source. The general ideas are very similar in all systems, though.

SASE is a CER engine which translates patterns into non-deterministic automata, acting as its computational model. As far as its input is concerned, it assumes that each

SDE is represented by an event type, a timestamp in the form of a single timepoint and any other extra attributes. All SDEs are also assumed to be in a single stream (in the case of multiple streams, these must first be merged into a single stream) and this stream is temporally ordered. A CE pattern is defined through a SQL-like language, whose purpose is to detect SDEs occurring in the specified temporal order and satisfying any extra constraints (possibly atemporal) in the form of predicates. The pattern of Figure 2.1 is an example of a SASE pattern as applied to the maritime domain. The PATTERN clause (line 1) captures a sequence of events, where the first indicates that a vessel starts moving at a low speed, then executes one or more turns and finally ends by a single event indicating that the slow movement has ended. The WHERE clause (lines 2–4) does three things: 1) determines the selection policy as skip-till-next-match; 2) partitions the stream according to the id of the vessels so that the pattern is applied to each vessel individually; 3) imposes an atemporal constraint on all the turn events so that the heading of each turn event differs by more than 90 degrees from the heading of the previous turn event. Finally, the WITHIN constraint (line 5) requires that the pattern happens within 21600 seconds (6 hours). This could be a simple pattern detecting zig-zag maneuvers, typical for vessels while trawling.

Figure 2.4 shows the automaton that would be constructed from this pattern. The PATTERN clause is first used to build the structure of the automaton and subsequently the WHERE clause determines the predicates that must be placed on the transitions. This automaton is non-deterministic as multiple outgoing edges from a state may evaluate to TRUE at the same time (upon reading the same event). Non-determinism essentially implies that, when an automaton can follow more than one edges, it must be cloned, i.e., a new run must be created, with each run following a different "trajectory" from that point on. When a run of an automaton follows a transition denoted by a solid line – see Figure 2.4 – the triggering event is stored in a buffer as being relevant for a possible future complete match. When a transition denoted by a dashed line is followed, the triggering event is considered as irrelevant and is not stored. The multiple runs created due to non-determinism (which can be present even with strict contiguity) and the buffers storing the relevant events constitute the main runtime bottlenecks. These bottlenecks become more pronounced when skip-till-any-match is used and when the pattern includes iteration operators [143].

SASE can accommodate all operators (even negation) mentioned in Section 2.1.1 and all selection policies mentioned in Section 2.1.2. On the other hand, its operators are not fully compositional, e.g., nesting of iterations is not allowed. In theory, it also allows for event hierarchies through another clause, called RETURN, appended at the end of patterns and acting as a projection operator. However, its publicly available source code does not include this functionality [123]. As far as the allowed consumption policies are concerned, only reuse is available. Finally, the windows in SASE are strictly time-based and constitute logical views.

FlinkCEP [15] is close in spirit to SASE. It also employs non-deterministic automata, equipped with predicates on their transitions. From a language perspective, one important difference to SASE is that, strictly speaking, it does not have a language for defining patterns. Instead, the user is required to write a pattern in Java or Scala, as shown in Listing 2.1, where we define a simple pattern with a vessel starting from an idle status (speed less than 0.1 knots) and then accelerates to more than 15 knots within less than 10 seconds. Writing patterns in such a way is cumbersome and error-prone. On the other hand, the advantage of FlinkCEP is that patterns are compositional. For example, the *idle* pattern in the Listing 2.1 can be used to define other patterns as well, besides the *abruptStart* pattern.

FlinkCEP also accommodates all selection policies (and supports some more, see

Listing 2.1: Example of Scala source code for defining patterns in FlinkCEP

```
// initial pattern to start the sequence
val idle: Pattern[Event, _] = Pattern.begin("idle").where(event => event.getSpeed < 0.1)

// strict contiguity
val highSpeed: Pattern[Event, _] = idle.next("highSpeed").where(event => event.getSpeed > 15)

// relaxed contiguity
val highSpeed: Pattern[Event, _] = idle.followdBy("highSpeed").where(event => event.getSpeed > 15)

// non-deterministic relaxed contiguity
val highSpeed: Pattern[Event, _] = idle.followedByAny("highSpeed").where(event => event.getSpeed >
    15)

// window
val abruptStart: Pattern[Event, _] = highSpeed.within(Time.seconds(10))
```

[15]), albeit with a slightly different terminology: relaxed-contiguity is the equivalent of skip-till-next-match and non-deterministic-relaxed-contiguity the equivalent of skip-till-any-match. However, the semantics of its selection policies do not exactly correspond to those of SASE. For example, it seems that non-determinism is not applicable for strict-contiguity in FlinkCEP, which is at odds with the semantics of this policy in SASE. Consider the pattern $a;b^*;b$ and the simple stream a_1, b_1, b_2. Even with strict-contiguity, this pattern would require a non-deterministic automaton for SASE. After the arrival of b_1, SASE would have two NFA runs: one that would terminate the pattern, move the run to its final state and complete the match, producing $M_1 = \{a_1, b_1\}$; and one that would treat b_1 as belonging to the iteration operator and that would reach its final state after b_2, thus producing another match, $M_2 = \{a_1, b_1, b_2\}$. On the other hand, FlinkCEP would detect only M_1. Inferring the precise semantics of FlinkCEP's selection policies is not trivial, as its internals are not documented and the documentation provides only informal explanations. With respect to consumption policies, reuse and consume are supported (along with some variations), but not bounded-reuse. As with selection policies, there is a difference in terminology: NO_SKIP is the equivalent of reuse and SKIP_PAST_LAST_EVENT the equivalent of consume.

FlinkCEP can use all operators of our algebra of Section 2.1.1. In fact, it includes several extensions of these operators. For example, quantifiers, as used in regular expressions, are also available, for imposing lower and/or upper bounds on the number of repetitions of an iteration operator. It is also quite flexible with respect to the windows that may be applied to a pattern. Both time-based and count-based windows are available, which can also be either sliding or tumbling. Finally, as in SASE, windows in FlinkCEP are also logical views, which means that SDEs are not buffered in batches (as, for example, in Spark streaming [16]), but they are directly forwarded to the operators of a pattern.

Siddhi [127] is a commercial CER engine with capabilities similar to those of FlinkCEP. It is also based on state machines for pattern matching and can support all operators of our algebra and most of the selection and consumption policies. Contrary to FlinkCEP, Siddhi offers a language for defining patterns.

Cayuga [45, 46] is similar to SASE, but a bit earlier and relatively less expressive. It does not support windows, but supports iteration, although, like SASE, does not allow for nested iterations. Due to the absence of windows, avoiding unbounded storage in the presence of iteration is achieved by using an automaton model that stores only the attribute values of the most recent iteration. Cayuga uses skip-till-any-match as its selection and reuse as its consumption policy. As opposed to the systems presented thus far, in order to avoid semantical ambiguities arising when timepoints are used as timestamps [61], Cayuga treats events as durative, with instantaneous events also available as a special case. A sequence operator in Cayuga is satisfied if the involved events are not overlapping, i.e., the end timepoint of an event is smaller than the start timepoint of the next event in the sequence.

NextCEP [124] and DistCED [115] are two other automata-based systems. From a language perspective, they are very similar to Cayuga. They also treat events as durative in order to avoid the semantical issues mentioned above. Their focus is not so much on providing a language with more expressive power, but on optimizing the evaluation of automata for efficient, distributed processing, e.g., by query rewriting.

2.3 Logic–based systems

Besides automata-based systems, there exists another significant line of work where a CER system employs a logic-based temporal formalism (see [19] for a survey). In this case, patterns often have the form of a rule, with a head and a body defining the conditions which, if satisfied, lead to the detection of a CE. The semantics are those of the temporal formalism employed. The underlying mechanism for performing inference can vary: from Selective Linear Definite (SLD) resolution used in Prolog-based systems to directed graphs (resembling automata) constructed from the rules.

The Chronicle Recognition System (CRS) is an example of the latter case [50, 51, 52, 62]. Although it does not use the terminology typically encountered in CER, its concepts, methods and algorithms are very close in spirit. A chronicle in CRS terminology is essentially a CE, i.e., a set of events linked together by time constraints and whose occurrence may depend on the context. A chronicle definition resembles a logical rule, having a body and a head. Pattern (2.2) below presents a simplified definition of the *abruptStart* pattern in the language of CRS.

$$\begin{array}{ll}
1 & \text{chronicle } \mathit{abruptStart}[?\mathit{VesselId}](T_2)\ \{ \\
2 & \quad \text{event}(\mathit{AISMessage}[?\mathit{VesselId},\mathit{idle}],T_1) \\
3 & \quad \text{event}(\mathit{AISMessage}[?\mathit{VesselId},\mathit{highSpeed}],T_2) \\
4 & \quad T_2 - T_1 \text{ in } [1,10] \\
5 & \}
\end{array} \tag{2.2}$$

Variables start with upper case letters while predicates and constants start with lower case letters, as in logic programming. Prefixing a variable with ? denotes that it is an atemporal variable. A feature of CRS is that a CE, like *abruptStart*, may be defined through multiple rules, thus expressing disjunction. Another feature that CRS supports and is generally lacking in other CER systems is the fact that the subevents in the definition of a CE need not necessarily be totally ordered. For example, line 4 in Pattern (2.2) could be $T_2 - T_1$ in $[-4,6]$, indicating that the *highSpeed* event could precede the *idle* by at most 4 timepoints. CRS also supports negation and iteration, although an iteration operator must have explicit lower and upper bounds on the number of repetitions. On the other hand, mathematical operators are not allowed as constraints, e.g., stating that the speed of a vessel should be above or below a certain threshold, e.g., as we did in the FlinkCEP pattern shown in Listing 2.1. Such information must be provided explicitly to the system through preprocessing. Notice, for example, that in lines 2–3 of Pattern 2.2, we assume that there already exists an attribute in each SDE concerning the speed status of a vessel (whether it is idle or has a high speed).

In order to be evaluated, a CRS pattern is compiled to a Temporal Constraint Network (TCN), i.e., a graph whose nodes correspond to events and edges encode the temporal constraints. This allows CRS to perform both consistency checking on the patterns and apply optimizations by propagating constraints in the graph or even removing them completely if it detects that they are redundant. It is also possible to provide semantics for the CRS language by using colored Petri Nets [35]. The runtime behavior of the CRS system is similar to that of automata-based systems, as instances of a TCN are continuously created and killed, according to whether future events can or cannot satisfy their constraints. skip-till-any-match and reuse are the default selection policy and consumption policy during evaluation, a fact which can lead to a substantial number of TCN instances being created and maintained. Although CRS does not directly use the notion of selection and consumption policies, it enlists techniques for reducing the number of active TCN instances which are closely related. A pattern may be required to detect chronicles that are not overlapping, a requirement which is a stricter variation of skip-till-next-match. Moreover, two TCN instances of the same pattern may also be forbidden to share events, which essentially corresponds to the consume consumption policy. CRS includes various other optimization techniques, such as "temporal focusing", which re-orders the states of the TCN based on event frequency; such techniques are being used in various contemporary CER approaches [81, 124].

We conclude this section by presenting RTEC (Event Calculus for Run-Time reasoning) [18], a CER engine based on the Event Calculus [82], written in Prolog. The Event Calculus is a logic programming action language that allows for reasoning about events and their effects. RTEC is an implementation of the Event Calculus tailored to event streams, by incorporating a windowing mechanism along with caching and indexing techniques for efficient reasoning. Patterns in RTEC are (locally) stratified logic programs [116]. RTEC patterns are usually written through initiatedAt and terminatedAt rules which determine when a CE starts and ceases to hold respectively. Pattern (2.3) is an example of a RTEC rule defining the *withinArea* CE, with which we want to detect intervals during which vessels are inside areas of a specific type, assuming that we also have SDEs (or lower level CEs) about the entrance and exit of vessels in and out of areas.

$$\begin{aligned}
&\mathsf{initiatedAt}(\mathit{withinArea}(\mathit{VesselId}) = \mathit{AreaType},\ T) \leftarrow \\
&\qquad \mathsf{happensAt}(\mathit{entersArea}(\mathit{VesselId},\mathit{Area}),\ T), \\
&\qquad \mathit{typeOf}(\mathit{Area},\mathit{AreaType}). \\
&\mathsf{terminatedAt}(\mathit{withinArea}(\mathit{VesselId}) = \_,\ T) \leftarrow \\
&\qquad \mathsf{happensAt}(\mathit{exitsArea}(\mathit{VesselId},\mathit{Area}),\ T).
\end{aligned} \tag{2.3}$$

RTEC will find all timepoints within a window in which *withinArea* is initiated, then compute the timepoints in which it is terminated, and finally calculate its maximal intervals by pairing initiating and terminating points. In other words, RTEC assumes that composite activities are subject to the law of inertia , i.e., a CE/fluent continues to hold unless explicitly terminated. Note also that RTEC makes no assumptions on the temporal distance between initiating and terminating points; these may be in different windows.

As is the case with CRS, a CE can have multiple definitions to indicate disjunction. It is also possible to define arbitrary temporal constraints among the timestamps of the subevents of a CE. Typically though, a global actual window (not a logical view) for all patterns is defined in order to restrict the search space of events. Contrary to CRS, besides temporal constraints, it is also possible to include mathematical operators. As a matter of fact, RTEC inherits the full expressive power of logic programming, and thus can naturally handle, among others, arbitrary constraints, relational CEs, and background knowledge. Another feature of RTEC is that it can handle both point-based and interval-based events and has constructs to manipulate time intervals, e.g., through union, intersection and complement.

RTEC does not include an explicit sequence operator, although sequencing can be defined through direct temporal constraints. However, due to the fact that Prolog's SLD-resolution is used for inference, all pairs of *AISMessage* events within the global window need to be checked and only those satisfying the temporal constraint are retained. It also does not support iteration, either unbounded or bounded. As is usual with most other CER systems, reuse is the only supported consumption policy. With respect to selection policies, there is a divergence between instantaneous and durative CEs. Instantaneous CEs follow skip-till-any-match whereas durative CEs follow a (deterministic) version of skip-till-next-match, since RTEC has been designed to compute the *maximal* intervals in which a CE is said to take place.

We also need to mention that there are some other logic-based CER systems. For example, PADRES is a distributed publish/subscribe messaging system [85] with fault detection and load balancing capabilities. It supports composite subscriptions (i.e., CEs) by combining low-level events through temporal and logical constraints. PADRES is based on Jess, a rule-based matching engine [77].

2.4 Tree–based systems

Another line of work on CER is the one that employs trees as a computational model, with ZStream being the prime example [98]. From a language perspective, it is interesting to note that ZStream is very similar to SASE, following the same syntax in most respects. For example, the fishing pattern of Figure 2.1 would be written in ZStream in an almost identical manner, with the exception of the clause for the selection policy, which would be absent. It is not easy to infer which selection policy ZStream follows, since this information is not directly reported, but we deduce from the provided examples that it must be skip-till-next-match. Thus, the main contribution of ZStream does not lie in offering more expressive power, but in using trees as the underlying computational model, which opens up avenues for optimizations.

ZStream differs from automata-based models in that it assumes that CEs are durative. A sequence operator among CEs is satisfied if the end timepoint of one CE is smaller than the start timepoint of the next CE in the sequence. This allows ZStream to avoid semantical ambiguities when hierarchies of events are present. For example, if $R_1 := a;b$ is a pattern with a window W_1, then each CE detected would typically acquire the timestamp of the last event, in this case of b. Now, if we define another pattern as $R_2 := R_1;c$ with a window W_2, then the intended semantics should be that R_2 is equivalent to $a;b;c$ with the extra constraints $b.timestamp - a.timestamp < W_1$ and $c.timestamp - a.timestamp < W_2$. However, the expression $R_2 := R_1;c$ could be translated to an automaton violating the second window constraint, since R_1 would have the timestamp of b and not of a. By treating CEs of R_1 as durative, ZStream can avoid such issues.

ZStream translates patterns to trees, whose leaves store SDEs and internal nodes correspond to operators. At the same time, it does not eagerly evaluate a pattern's predicates. Instead, it first collects SDEs into batches and then starts the evaluation. The combination of trees and batch evaluation allows ZStream to follow various physical plans for a given pattern, according to the expected cost of its operators and predicates. Such optimizations could be applied even to the simplest of patterns. For example, for the pattern $a;b$, SASE would create a new NFA run for every a appearing in the stream, even if we knew that b is a rare event. On the other hand, ZStream can follow another plan, by waiting until a b event has arrived and then checking the previously arrived a events that fall within the specified window and simply discarding those that have "expired".

E-Cube [90] is another CER system which employs tree structures, albeit in a different manner than ZStream. With respect to its expressive power, E-Cube supports most of the common CER operators, with the exception of iteration. Its temporal model is based on intervals, with only the SDEs being instantaneous. Its selection and consumption policies cannot be unambiguously determined in the work presenting it, but, by the definition of the sequence operator, we deduce that it probably follows skip-till-any-match and reuse.

The main power of E-Cube lies in its capabilities of multi-query optimizations, i.e., in its ability to evaluate multiple patterns while avoiding redundant and duplicate computations when the patterns under evaluation share some common structure. It achieves this by constructing an event pattern query hierarchy that allows for sharing of subpatterns, thus eliminating redundancy. Additionally, it employs a cost-driven optimizer that can devise an optimal plan in the sense of finding the plan that allows for maximal re-use of intermediate results. Importantly, E-Cube also has elastic properties. It continuously gathers stream statistics and can adapt online to a new execution plan when a drift is detected.

2.5 Hybrid approaches

ZStream is a purely tree-based CER engine. Esper also uses trees for the core of its functionality, like filtering, windowing and aggregations [54]. However, Esper is not easy to classify, since it also uses non-deterministic automata for its "match-recognize" pattern matching functionality, i.e., for the regular part of a pattern. Allen's interval algebra is also used for some of its time methods. This mixture of trees, automata and logic makes Esper patterns quite expressive, but the consequences on the semantics, soundness and completeness are unclear.

The T-Rex system, using TESLA as an event specification language, is an example of a CER system combining logic-based rules with automata [40, 41]. T-REX represents a transitional system from automata to logic, as it is not purely logic-based. Its patterns, written in TESLA, have a syntax similar to SASE and they are also translated to automata in order to be evaluated upon an event stream. However, TESLA patterns can be translated to TRIO formulas [63], i.e., to formulas of a first-order logical language that supports temporal operators and has clear semantics in terms of a metric temporal logic. Pattern (2.4) is an example of a TESLA pattern, capturing the FlinkCEP pattern presented in Listing 2.1, where we assume that the *Idle* and *HighSpeed* CEs have already been defined by other patterns. The sequence operator is defined using the WITHIN clause: it states that a *HighSpeed* event must follow an *Idle* event in no more than 10 seconds. Partitioning (by *VesselId*) is denoted through the use of the $ operator.

$$\begin{array}{l}
1\ \textsf{DEFINE}\ \mathit{AbruptStart}() \\
2\ \textsf{FROM}\ \mathit{Idle}(\mathit{VesselId} = \$x) \\
3\ \textsf{AND}\ \mathit{last\ HighSpeed}(\mathit{VesselId} = \$x) \\
4\ \textsf{WITHIN}\ 10\mathit{sec\ from\ Idle} \\
5\ \textsf{CONSUMING}\ \mathit{Idle}
\end{array} \tag{2.4}$$

TESLA supports most CER operators except disjunction. It also supports hierarchies of events. With respect to selection policies, it can define patterns with skip-till-any-match, but also makes some other policies available, similar to skip-till-next-match. For example, if we had a stream with two *Idle* events and one *HighSpeed*, then Pattern (2.4) would select only the second *Idle*, as denoted by the *last* keyword in line 3. By using the *first* keyword, it would select only the first *Idle* event. Notice that TESLA is one of the few systems that can define different consumption policies. Line 5 in Pattern (2.4) imposes the consume policy on the pattern.

2.6 Open issues & critical discussion

We conclude our treatment of CER languages with a critical discussion that attempts to pinpoint the weaknesses of the various approaches. For easier reference, we also provide an overview of the discussed systems in Table 2.1. A question mark in a cell indicates that there are not enough details in the paper(s) describing the relevant system to draw conclusions about the operator/functionality corresponding to the cell's column.

According to our discussion thus far, automata-based CER systems seem to satisfy many of the requirements described in Section 2.1.5. Especially solutions, such as FlinkCEP, offer a significant number of extra features, thus providing substantial flexibility for defining patterns. A closer look, however, reveals that there still exists a number of pending issues.

The existence of multiple systems, each with its own language and its own variation of automaton model, might be viewed as a sign of vigor for the field. On the other hand, this heterogeneity, where the semantics of a language have to be inferred from the operational semantics of the employed automata, can be a source of confusion. A discussion about the formal semantics of automata might seem like a purely theoretical endeavor, but the lack of such semantics can have important implications. Consider the requirement for event hierarchies and compositional patterns. It is a well-known fact that classical regular expressions and automata have nice closure properties, thus allowing for compositional definitions of expressions [76]. Interestingly, this is also the case for symbolic automata, i.e., automata that have predicates on their transitions and resemble the automaton models proposed for CER [44]. However, by adding memory to such automata, so that predicates relating more than one event are possible, and the need for marking SDEs as being part of match, we essentially move to symbolic transducers with memory, in which case some of the closure properties start breaking down [43, 79]. Note, for example, that classical regular expressions and automata are closed under iteration, i.e., we can take any regular expression/automaton, apply an iteration operator and the result will still be a regular expression/automaton. This is a procedure that can be repeatedly applied, thus allowing for nested iterations. On the other hand, both SASE and Cayuga construct acyclic automata allowing only self-loops on states to handle the operator of iteration, but not loops on the whole automaton (i.e., transitions from its final to its start state). This indicates that iteration cannot be arbitrarily used, as in regular expressions. It is therefore unclear which operators for defining a CER pattern may indeed be used compositionally and whether, if a pattern does make use of nested operators, its semantics will be as expected by a user (for a more detailed discussion of this issue, see [67]).

A related issue is that of the semantics of selection and consumption policies. On the one hand, consumption policies are often ignored, where reuse is implicitly the default policy, and their definition, whenever provided, is informal [56]. On the other hand, there is

System	σ	π	∨	∧	¬	;	*	W	H	T.M.	B.K.	S.P.	C.P.	Remarks
Automata														
SASE	✓	✓	✗	✗	✓	✓	✗	Logical	✗	Points	✗	all	Re	∨, ∧ and hierarchies possible in principle but not available in source code.
SASE+	✓	✓	✗	✗	✓	✓	✓	Logical	✗	Points	✗	all	Re	Iteration cannot be nested. ∨, ∧ and hierarchies possible in principle but not available in source code.
Cayuga	✓	✓	✓	?	✗	✓	✓	No windows	?	Intervals	✗	Stam	Re	
FlinkCEP	✓	✓	✓	?	✓	✓	✓	Logical	✓	Points	✗	all	Co,Re	Patterns in Java or Scala. Extra selection and consumption policies available. Quantified iteration available.
NextCEP	✓	✗	✓	✗	✓	✓	✓	Logical	✗	Intervals	✗	Stnm variant	Re	Durative events to handle associativity of sequence operators.
DistCED	✓	✗	✓	✓	✓	✓	✓	Logical	✗	Intervals	✗	?	?	Sequence and concatenation have different semantics. Sequence does not allow sub-events to be overlapping, permissible in concatenation.
Siddhi	✓	✓	✓	✓	✓	✓	✓	Logical	✓	Points	✗	Sc,Stam	Co,Re	Quantified iteration available.
Logic														
CRS	✓	✗	✓	?	✓	✓	✓	Logical	✓	Points	✗	Stam, Stnm variant	Co,Re	Only equality predicates (unification) for σ. Iteration explicitly bounded.
RTEC	✓	✓	✓	✓	✓	✓	✗	Actual	✓	Points + Intervals	✓	Stam for instantaneous CEs, Stnm variant for durative	Re	Sequence through explicit time constraints.
PADRES	✓	?	✓	✓	?	✓	✓	?	✓	Points	✗	?	?	Based on Jess [77]. Iteration explicitly bounded.
Trees														
ZStream	✓	✓	✓	?	✓	✓	✓	Actual	✓	Intervals	✗	Stnm variant	Re	Reordered execution and lazy evaluation of patterns.
E-Cube	✓	✓	✓	✓	✓	✓	✗	Logical	✓	Intervals	✗	Stam	Re	Pattern hierarchies to share intermediate results.
Hybrid														
TESLA	✓	✓	✗	?	✓	✓	✓	Logical	✓	Points	✗	Stam, Stnm variant	Co,Re	Rules translatable to temporal logical formulas, converted to automata for evaluation.
Esper	✓	✓	✓	?	✓	✓	✓	✓	✓	Points	✗	?	?	Mixture of trees, automata and Allen's interval algebra. Windows available, but type unknown.
System	σ	π	∨	∧	¬	;	*	W	H	T.M.	B.K.	S.P.	C.P.	Remarks

Table 2.1: Expressive capabilities of CER systems.
σ: selection, π: projection, $\wedge$: conjunction, $\vee$: disjunction, $\neg$: negation, ;: sequence, *: iteration, W: windowing, H: hierarchies, T.M.: temporal model, B.K.: background knowledge, S.P.: selection policies, C.P.: consumption policies, Stam : skip-till-any-match, Stnm : skip-till-next-match, Sc : strict-contiguity, Co : consume, Re : reuse.

no consensus regarding the semantics of selection policies. Even at the terminology level, there is substantial divergence, where certain policies, like partition-contiguity, might be subsumed under a different notion (see again Section 2.1.2). As a final note with respect to the issues of semantics, we would like to point out that the lack of well-defined semantics can be an obstacle to applying certain optimization techniques, like query rewriting, since these require a methodology for determining when two queries are in fact equivalent.

From the point of view of expressiveness, automata are, as expected, well-suited to the detection of sequential patterns. There is no CER system that is based on automata and can handle concurrency though. A conceivable solution would be to define patterns with concurrent events by using (in)equality predicates on the timestamps of SDEs. Such a solution would, however, defeat the purpose of using automata and could possibly complicate their semantics even further. Automata assume that their input symbols arrive at a certain order, which, in CER, is the temporal order of the SDEs. This simple temporal model also allows the detected CEs to be temporally ordered. Now, assume, as an example, that we need to detect two concurrent durative SDEs by using a pattern like b during a, meaning that b must happen while a is happening, i.e., $b.start > a.start$ and $b.end < a.end$ (as in Allen's interval algebra [12, 13]). If there in fact exist two such SDEs in a stream, the first issue is the order in which they should be presented to an automaton. An option would be to present b first, since this is the event for which we first know all the information we need, i.e., both its start and end timepoints. Thus, upon reading b the automaton could move to a next state. Upon reading a, it could check the inequality constraint regarding the start timepoints (the constraint about the end timepoints can be skipped since we assumed SDEs are presented according to their end timepoints) and then produce a new CE. The end timepoint of this new CE though would have to be equal to $b.end$. As a result, we have produced a new event whose end timepoint is actually behind the last end timepoint of our stream, that of a. Meanwhile, any other events happening between $b.end$ and $a.end$ might have already been processed by the engine, without taking into account our new CE. A way out of this conundrum would be to send the start and end timepoints of each event separately, e.g., $a.start, b.start, b.end, a.end$. This solution could work for SDEs, but not for CEs, since, for a CE, we cannot always know when it started until we have actually seen the last event of its sequence. For example, a WITHIN constraint forces us to wait for the last event in order to make sure that the time difference between the last and the first event is less than that imposed by it. This simple example illustrates a possible difficulty in handling concurrent events using automata; other formalisms, such as Petri Nets [100] which are often used for modeling concurrent processes, could possibly prove helpful.

The lack of a mechanism for incorporating background knowledge in automata-based CER systems is another obstacle for defining yet more expressive patterns. This does not seem to be a serious limitation though; [8] is an example of a system that employs automata with predicates drawn from a knowledge base. Since these types of automata have predicates which can be Boolean formulas, it is possible to use a solver underneath, providing not only logical facts, but also further (domain-dependent) axioms, along with a full-fledged inference engine. This is indeed the approach followed in the line of work concerning symbolic automata [134]. As a downside to this increased expressiveness, it is worth noting that performing heavy inference tasks online could significantly affect the performance of a CER system. It is also interesting to note the lack of support for conjunctive (involving $\wedge$) patterns in most automata-based methods, although conjunction can often be a useful operator. Conjunctive patterns are generally more expensive than strictly sequential ones since the ordering constraint is lifted and more matches tend to be produced. In [48], optimization techniques for handling such conjunctive patterns are presented, based on static and runtime unsatisfiability checking.

As far as the logic-based approaches are concerned, the issues regarding their semantics may not be so pronounced, but are not absent. For example, CRS may be described through the semantics of Petri nets, but this is still far from a purely declarative semantics. On the other hand, RTEC has declarative semantics, but the issue of ambiguities concerning instantaneous events (see the discussion in Section 2.4) may still arise, if caution is not exercised. With respect to TESLA, its rules follow the semantics of a temporal formalism (TRIO), but uses automata as its underlying computational model. However, it is not proven that the algorithm for constructing TESLA automata does indeed produce automata whose semantics are equivalent to those of TRIO formulas. Proving such an equivalence is not trivial, since, as already mentioned, extended versions of automata do not always have the compositional properties taken for granted in logical formulas. Please, also note that, even among logic-based systems, the semantics show great variation, from automata (TESLA) to Petri nets (CRS) and logic programming (RTEC), thus still lacking homogeneity.

Compared to automata, logic-based systems also seem to have a somewhat different focus with respect to their expressiveness. They can naturally express concurrency, as in CRS and RTEC, and to support hierarchies of events. On the other hand, iteration is either not supported (RTEC) or needs to be explicitly bounded (CRS). This also implies that they do not also support complex mathematical operators, like aggregates (e.g., averages, maximum/minimum values) that need to be applied to all the events selected by an iteration operator. In fact, CRS does not support any such operators, even without iteration. Given that engines for logic inference can easily incorporate knowledge in the form of facts, it is surprising that only RTEC can actually take advantage of any background knowledge. Except for TESLA, they also tend to ignore the existence of the various selection and consumption policies. Finally, with respect to performance, logic-based approaches have proven at least as efficient as automata and tree-based approaches.

Tree-based approaches provide multiple physical plans, either for a single pattern (ZStream) or for multiple patterns (E-Cube), which, in turn, allows for a more efficient pattern evaluation, according to the cost of each plan. Ideas similar to those of reordered execution and lazy evaluation behind ZStream have also been applied to automata-based systems [81, 124].

We would also like to make some remarks concerning the requirements for relational events and patterns without windows (see Section 2.1.5). With the exception of Cayuga, all other systems require windows in order to function. Cayuga can handle windowless patterns by limiting its expressive power. RTEC also relaxes to an extent the requirement for windows, at least for durative CEs. Durative CEs in RTEC need not have both their initiating and terminating timepoints within the same window. This is manageable in RTEC due to the fact that durative CEs are evaluated according to a deterministic version of the skip-till-next-match policy, i.e., any new initiation timepoints after the initial initiation do not result in new CE candidates being created. It is thus an open issue how to handle windowless patterns in the presence of relaxed policies. With respect to relational events, the issue is not that of a lack of expressive power. Even automata-based systems could handle such patterns, through a constraint imposing that the object identifiers of two successive events are different; doing, in a sense, the opposite of what partition-contiguity does. Efficiency is the real issue in this case, especially when the CE has high arity (more than two objects need to be related), as such patterns would result in a significant increase in the number of created runs for automata-based systems and in a more expensive searching process for logic-based systems.

Another area that has not received significant attention in CER is that of automatically learning a set of patterns from the stream of input SDEs. One example of a method for the automatic extraction of CER patterns may be found in [80], where a parallel system for learning theories in the form of Event Calculus rules (as in RTEC) is presented. The issue of parallelization will be discussed in the following sections. Other examples for learning CER patterns are [27, 84, 96, 126].

Summarizing our discussion, the most obvious research gap may be the lack of a common formal framework and of a universally agreed terminology, also indicated by the fact that there still do not exist any standard CER benchmarks. Automata-based methods seem to support most of the core CER operators, but it is still unclear how these operators may be used and what their semantics should be. It also remains an open issue how the common case of relational events could be supported. On the other hand, logic-based methods have clearer semantics, but they do not support all operators, such as iteration (or support a limited version), while they are also less flexible with respect to the allowed selection policies. The absence of support for background knowledge and the inability to handle both instantaneous and durative events are also worth noting for most methods.

3. Literature review

There are multiple ways to define the task of forecasting over time-evolving data streams. Before proceeding with the presentation of previous work on forecasting, we first begin with a terminological clarification. It is often the case that the terms "forecasting" and "prediction" are used interchangeably as equivalent terms. For reasons of clarity, we opt for the term of "forecasting" to describe our work, since there does exist a conceptual difference between forecasting and prediction, as the latter term is understood in machine learning. In machine learning, the goal is to "predict" the output of a function on previously unseen input data. The input data need not necessarily have a temporal dimension and the term "prediction" refers to the output of the learned function on a new data point. For this reason we avoid using the term "prediction". Instead, we choose the term "forecasting" to define the task of predicting the temporally future output of some function or the occurrence of an event. Time is thus a crucial component for forecasting. Moreover, an important challenge stems from the fact that, from the (current) timepoint where a forecast is produced until the (future) timepoint for which we try to make a forecast, no data is available. A forecasting system must (implicitly or explicitly) fill in this data gap in order to produce a forecast, whereas in a typical machine learning task a prediction can be made using all available data that refer to the target point of the prediction.

In what follows, we present previous work on CEF, as defined above, in order of increasing relevance to CER. Since work on CEF has been limited thus far, we start by briefly mentioning some forecasting ideas from other fields and discuss how CEF differs from these research areas.

3.1 Time-series forecasting

Time-series forecasting is an area with some similarities to CEF and a significant history of contributions [99]. However, it is not possible to directly apply techniques from time-series forecasting to CEF. Time-series forecasting typically focuses on streams of (mostly) real-valued variables and the goal is to forecast relatively simple patterns. On the contrary, in CEF we are also interested in categorical values, related through complex patterns and involving multiple variables.

Another limitation of time-series forecasting methods is that they do not provide a language with which we can define complex patterns, but simply try to forecast the next value(s) from the input stream/series. In CER, the equivalent task would be to forecast the next input event(s) (SDEs). This task in itself is not very useful for CER though, since the majority of SDE instances should be ignored and do not contribute to the detection of CEs. For example, if we want to determine whether a ship is following a set of pre-determined waypoints at sea, we are only interested in the messages where the ship "touches" each waypoint. All other intermediate messages are to be discarded and should not constitute part of the match. CEs are more like "anomalies" and their number is typically orders of magnitude lower than the number of SDEs.

One could possibly try to leverage techniques from SDE forecasting to perform CE forecasting. At every timepoint, we could try to estimate the most probable sequence of future SDEs, then perform recognition on this future stream of SDEs and check whether any future CEs are detected. We have experimentally observed that such an approach yields sub-optimal results. It almost always fails to detect any future CEs. This behavior is due to the fact that CEs are rare. As a result, projecting the input stream into the future creates a "path" with high probability but fails to include the rare "paths" that lead to a CE detection. Because of this serious under-performance of this method, we do not present detailed experimental results.

3.2 Sequence prediction (compression)

Another related field is that of prediction of discrete sequences over finite alphabets and is closely related to the field of compression, as any compression algorithm can be used for prediction and vice versa. The relevant literature is extensive. Here we focus on a sub-field with high importance for our work, as we have borrowed ideas from it. It is the field of sequence prediction via variable-order Markov models [24, 28, 37, 120, 121, 140]. As the name suggests, the goal is to perform prediction by using a high-order Markov model. Doing so in a straightforward manner, by constructing a high-order Markov chain with all its possible states, is prohibitively expensive due to the combinatorial explosion of the number of states.

Variable-order Markov models address this issue by retaining only those states that are "informative" enough. In Section 6.2.2, we discuss the relevant literature in more details. The main limitation of previous methods for sequence prediction is that they they also do not provide a language for patterns and focus exclusively on next symbol prediction, i.e., they try to forecast the next symbol(s) in a stream/string of discrete symbols. As already discussed, this is a serious limitation for CER.

An additional limitation is that they work on single-variable discrete sequences of symbols, whereas CER systems consume streams of events, i.e., streams of tuples with multiple variables, both numerical and categorical. Notwithstanding these limitations, we show that variable-order models can be combined with symbolic automata in order to overcome their restrictions and perform CEF.

3.3 Temporal mining

A significant number of forecasting methods comes from the field of temporal pattern mining, where patterns are usually defined either as association rules [4] or as frequent episodes [95].

For example, in [135], a framework similar to that of association rule mining is used in order to identify sets of event types that frequently precede a rare, target event within a temporal window. However, the goal is not restricted to identifying frequent itemsets, but rather to be able to identify sets of event types that frequently precede a rare, target event within a temporal window.

In [83], a probabilistic model is presented. The goal is to calculate the probability of the immediately next event in the stream through a combination of standard frequent episode discovery algorithms, Hidden Markov Models and mixture models. First, through a standard frequent episode discovery algorithm, significant episodes that (immediately) precede the target events are extracted. Each such episode is then used to build a set of Hidden Markov Models (HMM), which are finally combined into a mixture model through an Expectation Maximization procedure.

Episode rules constitute the framework of [57] as well, where the goal is to mine predictive rules whose antecedent is minimal (in number of events) and temporally distant from the consequent.

The algorithms presented in [144] focus on batch, online mining of sequential patterns, without maintaining exact frequency counts. At any time, the learned patterns (up to that time) can be used to test whether a prefix matches the last events seen in the stream and therefore make a forecast. This paper also includes an extensive discussion on how precision may be defined when dealing with forecasts, an important topic in itself, since it is not always obvious how usual metrics (like precision, recall and F_1 score) may be transferred to forecasting (e.g., multiple forecasts may be produced and revised for a single event).

The method proposed in [34] starts with a given episode rule (as a Directed Acyclic Graph) and The rule is in the form of a directed acyclic graph (DAG) for encoding the temporal order of the involved events. It is also accompanied by the time window within which the events in the antecedent must occur, by another window within which the consequent will occur and by a probability value for the occurrence of the consequent. the goal is to build appropriate data structures that can efficiently detect the minimal occurrences of the antecedent of a rule defining a complex event, i.e., those “clusters” of antecedent events that are closer together in time.

From the perspective of CER, the disadvantage of these methods is that they usually target simple patterns, defined either as strict sequences or as sets of input events. Moreover, the input stream is composed of symbols from a finite alphabet, as is the case with the compression methods mentioned above.

3.4 Sequence prediction based on neural networks

Lately, a significant body of work has focused on event sequence prediction and point-of-interest recommendations through the use of neural networks (see, for example, [32, 87]). These methods are powerful in predicting the next input event(s) in a sequence of events, but they suffer from limitations already mentioned above. They do not provide a language for defining complex patterns among events and their focus is thus on SDE forecasting. An additional motivation for us to first try a statistical method rather than going directly to neural networks is that, in other related fields, such as time series forecasting, statistical methods have often been proven to be more accurate and less demanding in terms of computational resources than ML ones [94].

3.5 Process mining

Compared to the previous categories for forecasting, the field of process mining is more closely related to CER [131]. Processes are typically defined as transition systems (e.g., automata or Petri nets) and are used to monitor a system, e.g., for conformance testing. Process mining attempts to automatically learn a process from a set of traces, i.e., a set of activity logs. Since 2010, a significant body of work has appeared, targeting process prediction, where the goal is to forecast if and when a process is expected to be completed (for surveys, see [58, 97]). According to [97], until 2018, 39 papers in total have been published dealing with process prediction.

At a first glance, process prediction seems very similar to CEF. At a closer look though, some important differences emerge. An important difference is that processes are usually given directly as transition systems, whereas CER patterns are defined in a declarative manner. The transition systems defining processes are usually composed of long sequences of events. On the other hand, CER patterns are shorter, may involve Kleene-star, iteration operators (usually not present in processes) and may even be instantaneous. Consider, for example, a pattern for our running example, trying to detect speed violations by simply checking whether a vessel's speed exceeds some threshold. This pattern could be expanded to detect more violations by adding more disjuncts, e.g., for checking whether a vessel is sailing within a restricted area, all of which might be instantaneous. A CEF system cannot always rely on the memory implicitly encoded in a transition system and has to be able to learn the sequences of events that lead to a (possibly instantaneous) CE.

Another important difference is that process prediction focuses on traces, which are complete, full matches, whereas CER focuses on continuously evolving streams which may contain many irrelevant events. A learning method has to take into account the presence of these irrelevant events. In addition to that, since CEs are rare events, the datasets are highly imbalanced, with the vast majority of "labels" being negative (i.e., most forecasts should report that no CE is expected to occur, with very few being positive). A CEF system has to strike a fine balance between the positive and negative forecasts it produces in order to avoid drowning the positives in the flood of all the negatives and, at the same time, avoid over-producing positives that lead to false alarms. This is also an important issue for process prediction, but becomes critical for a CEF system, due to the imbalanced nature of the datasets.

3.6 Complex event forecasting

Contrary to process prediction, forecasting has not received much attention in the field of CER, although some conceptual proposals have acknowledged the need for CEF [36, 53, 60].

The first concrete attempt at CEF was presented in [101]. A variant of regular expressions was used to define CE patterns, which were then compiled into automata. These automata were translated to Markov chains through a direct mapping, where each automaton state was mapped to a Markov chain state. Frequency counters on the transitions were used to estimate the Markov chain's transition matrix. This Markov chain was finally used to estimate if a CE was expected to occur within some future window. As we explain in Section 6.2.2, in the worst case, such an approach assumes that all SDEs are independent (even when the states of the Markov chain are not independent) and is thus unable to encode higher-order dependencies. This issue is explained in more detail in Section 6.2.2.

Another example of event forecasting was presented in [5]. Using Support Vector Regression, the proposed method was able to predict the next input event(s) within some future window. This technique is similar to time-series forecasting, as it mainly targets the prediction of the (numerical) values of the attributes of the input (SDE) events (specifically, traffic speed and intensity from a traffic monitoring system). Strictly speaking, it cannot therefore be considered a CE forecasting method, but a SDE forecasting one. Nevertheless, the authors of [5] proposed the idea that these future SDEs may be used by a CER engine to detect future CEs. As we have already mentioned though, in our experiments, this idea has yielded poor results.

In [110], Hidden Markov Models (HMM) are used to construct a probabilistic model for the behavior of a transition system describing a CE. The observable variable of the HMM corresponds to the states of the transition system, i.e., an observation sequence of length l for the HMM consists of the sequence of states visited by the system after consuming l SDEs. These l SDEs are mapped to the hidden variable, i.e., the last l values of the hidden variable are the last l SDEs. In principle, HMMs are more powerful than Markov chains. In practice, however, HMMs are hard to train ([2, 24]) and require elaborate domain modeling, since mapping a CE pattern to a HMM is not straightforward (see Section 6.2.2 for details). In contrast, our approach constructs seamlessly a probabilistic model from a given CE pattern (declaratively defined).

Automata and Markov chains are again used in [6, 8]. The main difference of these methods compared to [101] is that they can accommodate higher-order dependencies by creating extra states for the automaton of a pattern. The method presented in [6] has two important limitations: first, it works only on discrete sequences of finite alphabets; second, the number of states required to encode long-term dependencies grows exponentially. The first issue was addressed in [8], where symbolic automata are used that can handle infinite alphabets. However, the problem of the exponential growth of the number of states still remains. We show how this problem can be addressed by using variable-order Markov models.

A different approach is followed in [86], where knowledge graphs are used to encode events and their timing relationships. Stochastic gradient descent is 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 SDE forecasting, as it does not target complex events. More precisely, it tries to forecast which predicates the forthcoming SDEs will satisfy, without taking into account relationships between the events themselves (e.g., through simple sequences).

3.7 Other

Timewewaver is a genetic algorithm that tries to learn from sequences of events a set of predictive patterns [139]. Its focus is on learning patterns that can forecast, as early as possible, rare events, such as equipment failures.

In [49], the forecasting problem is formulated as a classification problem and the goal is again to construct predictive patterns for rare events. The proposed algorithm constructs a matrix of features, finds a reduced set of features through Singular Value Decomposition and then trains a set of Support Vector Machines, one for each target event. The proposed algorithm uses a monitoring window in order to construct a matrix of features based on the event types present in this window. This matrix subsequently undergoes through singular-value decomposition (SVD) so that a reduced set of correlated features may be found. Finally, a set of support vector machines (SVM) is trained, one for each target event, using the set of reduced features.

In [142], a variant of decision trees is used in order to learn sequence prefixes that are as short as possible and that can forecast the class label of the whole sequence.

In [70], Piecewise-Constant Conditional Intensity Models and decision trees are employed in order to learn a very fine-grained model of the temporal dependencies among events in sequences. The learned models can then be used to calculate whether a sequence of target events will occur in a given order and in given time intervals.

One of the earliest methods for forecasting is the Chronicle Recognition System, proposed in [52, 62], where events may be associated with both temporal operators and with numerical constraints on their timestamps. The system uses partial matches in order to report when the remaining events are expected for the pattern to complete. However, such forecasts are not based on a confidence or probability metric.

II Part Two

4. Forecasting with pattern Markov chains

In this chapter we present an initial attempt at Complex Event Forecasting. We assume that event patterns are defined through (classical) regular expressions. As a first step, these patterns are converted to finite automata for the purpose of pattern matching. Subsequently, these automata are converted into Markov chains, which allow for the construction of a probabilistic model for the initial pattern. The final goal is to be able to forecast, as events arrive at the system, when the pattern will be fully matched. This is the first time that Pattern Markov Chains are used for online event forecasting. We show that our system can indeed forecast the completion of patterns in real-world datasets and that, under certain assumptions, it can do so with guaranteed precision. Moreover, we explore the quality of the produced forecasts, using three different metrics: precision score, spread, which refers to how focused a forecast is, and distance, which captures how early a forecast is reported.

The structure of the chapter is as follows: In Section 4.1, the necessary mathematical terminology and framework are described. Section 4.2 elaborates on the implementation details of the system, while Section 4.3 presents experimental results on real-world datasets. Finally, in Section 4.4 we conclude with a summary.

4.1 Theoretical background

The problem we address could be stated as follows. Given a stream of events S and a pattern R, the goal is two-fold. First: find the full matches of R in S. Second: as the stream is consumed by the engine, forecast the full matches before they are detected by the recognition engine. For the recognition task, we use finite automata, whereas for forecasting, we convert these automata into Markov chains.

We make the following assumptions:

- Patterns are defined in the form of regular expressions.
- The *selection strategy* is either that of *contiguity* (i.e., events in a match must be contiguous, without irrelevant events intervening) or *partition-contiguity* (i.e., same as *contiguity* but stream may be partitioned by a specific event attribute). The

counting policy is that of *non-overlap* (i.e., after a full match, the automaton returns to its start state). See [143] for the various selection strategies and [91] for the counting policies.
- The stream is generated by a m-order Markov process.
- The stream is stationary, i.e., its statistical properties remain the same. Hence the constructed Markov chain is *homogeneous* and its transition matrix remains the same at all time-points.
- A forecast reports for how many "points" we will have to wait until a full match. By the term "point", we refer to number of transitions of the Markov chain (or equivalently to number of future events) and not to time-points. Points are indeed time-points only in cases where a new event arrives at each time-point.

The theoretical tools presented in this section are based mostly on the work described in [59, 103, 107] and are grounded in the field of string pattern matching. For a review of this field, the reader may consult [91]. Comprehensive treatments of this subject may be found in [39, 71, 92].

4.1.1 Event recognition

In this section, we briefly review some of the necessary terminology [76]. Regular expressions define the so-called regular languages. Within the context of the theory of regular languages, an *alphabet* $\Sigma = \{e_1, ..., e_r\}$ is a finite, non-empty set of symbols. The alphabet essentially refers to the set of the different event types that may appear in the stream. A string over Σ is a finite sequence of symbols from the alphabet. A language L over Σ is a set of strings over Σ. One common way to denote languages over Σ is through the use of regular expressions. If R denotes a regular expression, then $L(R)$ denotes the language defined by R. There are three operators that are used in regular expressions: *union*, which is binary and is denoted by the symbol $+$, *concatenation*, again binary, denoted by $\cdot$, and *star closure*, which is unary, denoted by *. Regular expressions are inductively defined as follows

- The *union* of two languages L and M, $L \cup M$ is the set of strings that belong either to L or M. If R_1 and R_2 are regular expressions, then $R_1 + R_2$ is also a regular expression and $L(R_1 + R_2) = L(R_1) \cup L(R_2)$. Union corresponds to the *OR* operator in event recognition.
- The *concatenation* of two languages L and M, $L \cdot M$ is the set of strings formed by concatenating strings from L with strings from M, i.e., $L \cdot M = \{s_1 \cdot s_2, s_1 \in L, s_2 \in M\}$. If R_1 and R_2 are regular expressions, then $R_1 \cdot R_2$ is also a regular expression and $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$. Concatenation corresponds to the *sequence* operator in event recognition.
- The *star closure* of a language L is $L^* = \bigcup\limits_{i \geq 0} L^i$, where L^i is concatenation of L with itself i times. If R is a regular expression, then R^* is also a regular expression and $L(R^*) = (L(R))^*$. Star closure corresponds to the *iteration* operator in event recognition.

Finally, the inductive basis for a regular expression is that it may also be the empty string or a symbol from Σ.

Regular expressions may be encoded by deterministic and non-deterministic finite automata (DFA and NFA respectively).

Definition 4.1.1 — Deterministic Finite Automaton. A DFA is 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite set of symbols, $\delta : Q \times \Sigma \rightarrow Q$ a transition function from a state reading a single symbol to another state, $q_0 \in Q$ a start state and $F \subset Q$ a set of final states. A string $s = e_1 e_2 ... e_d \in \Sigma^*$ is accepted by the DFA if $\delta(q_0, s) \in F$, where the transition function for a string is defined as $\delta(q, e_1 e_2 ... e_d) = \delta(\delta(q, e_1 e_2 ... e_{d-1}), e_d)$.

The definition for a NFA is similar with the modification that the transition function is now $\delta : Q \times \Sigma \rightarrow S_Q$, where S_Q is the power set of Q.

There exist well-known algorithms for converting a regular expression R to an equivalent NFA, NFA_R, and subsequently to an equivalent DFA, DFA_R [76]. For event recognition, a slight modification is required so that the DFA can detect all the full matches in the stream. The regular expression and DFA that should be used are $\Sigma^* \cdot R$ and $DFA_{\Sigma^* \cdot R}$ respectively so that the DFA may recognize all the strings ending with R [39, 71, 103].

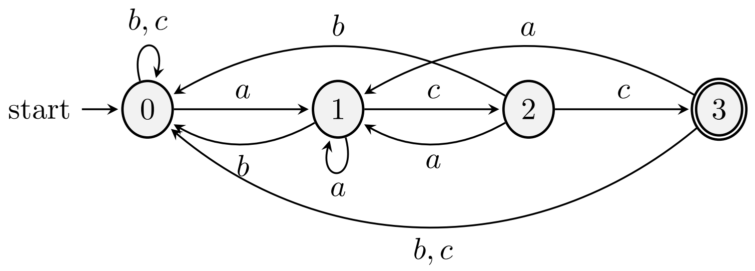
■ **Example 4.1** Figure 4.1a shows an example of a DFA, constructed for $R = a \cdot c \cdot c$ (one event of type a followed by two events of type c) and $\Sigma = \{a, b, c\}$ (three event types may be encountered, a,b and c). ■

4.1.2 Event forecasting

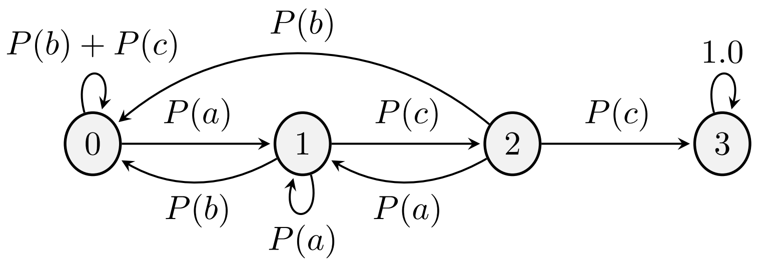
We use Pattern Markov Chains, i.e., convert $DFA_{\Sigma^* \cdot R}$ to an "appropriate" Markov chain. The Markov chain should be "appropriate" in the sense that it could be used in order to make probabilistic inferences about the run-time behavior of $DFA_{\Sigma^* \cdot R}$. In the case where the stream consumed by $DFA_{\Sigma^* \cdot R}$ is assumed to be composed of a sequence of independent, identically distributed (i.i.d.) events from Σ, then constructing the corresponding Markov chain is straightforward. As shown in [107], if $X = X_1, X_2, ..., X_i, ...$ is the i.i.d. sequence of input events, then the sequence $Y = Y_0, Y_1, ..., Y_i, ...$, where $Y_0 = q_0$ and $Y_i = \delta(Y_{i-1}, X_i)$ (i.e., the sequence of the states that $DFA_{\Sigma^* \cdot R}$ visits) is a 1-order Markov chain. Such a Markov chain, associated with a pattern R, is called a Pattern Markov Chain (PMC). Moreover, the transition probabilities between two states are simply given by the occurrence probabilities of the event types. If $p, q \in Q$ and Π is the $|Q| \times |Q|$ matrix holding these probabilities, then $\Pi(p,q) = P(X_i = e)$, if $\delta(p, e) = q$ (otherwise, it is 0). This means that we can directly map the states of $DFA_{\Sigma^* \cdot R}$ to the states of a PMC and for each edge of $DFA_{\Sigma^* \cdot R}$ labeled with $e \in \Sigma$, we can insert a transition in the PMC with probability $P(e)$ (assuming here stationarity, i.e., $P(X_i = e) = P(X_j = e), \forall i, j$).

■ **Example 4.2** As an example, Figure 4.1b shows the PMC constructed for $R = a \cdot c \cdot c$, based on the $DFA_{\Sigma^* \cdot R}$ of Figure 4.1a (the reason why state 3 has only a self-loop with probability 1.0 will be explained later). ■

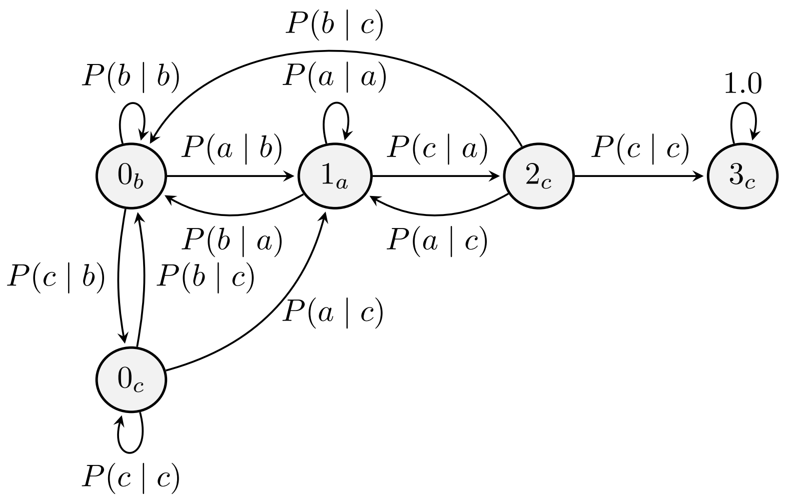
For the more general case where the process generating the stream is of a higher order $m \geq 1$, the states of the PMC should be able to remember the past m symbols so that the correct conditional probabilities may be assigned to its transitions. However, the states of $DFA_{\Sigma^* \cdot R}$ do not hold this information. As shown in [103, 107] we can overcome this problem by iteratively duplicating those states of $DFA_{\Sigma^* \cdot R}$ for which we cannot unambiguously determine the last m symbols that can lead to them and then convert it to a PMC. From now on, we will use the notation PMC_R^m to refer to the Pattern Markov Chain of a pattern R and order m. Please, note that, from a mathematical point of view, the resulting Markov chain is always of order 1, regardless of the value of m [107].



(a) $DFA_{\Sigma^* \cdot R}$.



(b) PMC^0_R.



(c) PMC^1_R.

Figure 4.1: DFA and PMCs for $R = a \cdot c \cdot c$, $\Sigma = \{a, b, c\}$ and for $m = 0$ and $m = 1$.

■ **Example 4.3** As an example, see Figure 4.1c which shows the resulting PMC^1_R for $R = a \cdot c \cdot c$. Note that the DFA for the same pattern with m=0, shown in Figure 4.1a, has a state which is ambiguous. When in state 0, the last symbol read may be either b or c. For all the other states, we know the symbol that led to them. Therefore, state 0 must be duplicated and state 0_c is added. Now, when in state 0_b, we know that the last symbol was b, whereas in state 0_c, it was c. ■

Once we have PMC^m_R for a user-defined pattern R, we may use the whole arsenal of Markov chain theory to make certain probabilistic inferences about R. For the task of forecasting, a useful distribution that can be calculated is the so-called waiting-time distribution. The waiting-time for a pattern R when its $DFA_{\Sigma^* \cdot R}$ is in state q is a random variable, denoted by $W_R(q)$. It is defined as the number of transitions until its first full match, i.e., until the DFA visits for the first time one of its final states.

$$W_R(q) = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\}$$

The DFA is in a non-final state q and we are interested in the smallest time index $n > 0$ (i.e., first time) at which it will visit a final state. Informally, what we want to achieve through $W_R(q)$ is the following: each time the DFA is in some non-final state q (regardless of whether it is the start state), we want to estimate how many transitions we will have to wait until it reaches one of its final states, i.e., until a full match is detected. This number of future transitions may be given to the user as a forecast and it is constantly revised as more symbols are consumed and the DFA moves to other states. As a random variable, $W_R(q)$ follows a probability distribution and our aim is to compute this distribution for every non-final state q.

We can compute the distribution of $W_R(q)$ through the following technique. First, we convert each state of PMC_R^m that corresponds to a final state f of $DFA_{\Sigma^* \cdot R}$ ($f \in F$, with $|F|$=k) into an absorbing state, i.e., a "sink" state with probability of staying in the same state equal to 1.0 (state 3 in Figure 4.1). We can then re-organize the transition matrix as follows:

$$\Pi = \begin{pmatrix} N & C \\ 0 & I \end{pmatrix} \tag{4.1}$$

where I is the identity matrix of size $k \times k$, corresponding to the absorbing states. If PMC_R^m has a total of l states (k of which are final), then N would be of size $(l-k) \times (l-k)$ and would correspond to the non-final states, holding the probabilities for all the possible transitions between (and only between) the non-final states. Finally, C is a $(l-k) \times k$ matrix holding the transition probabilities from non-final to final states and 0 is a zero matrix of size $k \times (l-k)$.

■ **Example 4.4** For example, for the PMC of Figure 4.1b, the transition matrix would be the following:

$$\Pi = \begin{Bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{Bmatrix} \begin{pmatrix} P(b)+P(c) & P(a) & 0 & 0 \\ P(b) & P(a) & P(c) & 0 \\ P(b) & P(a) & 0 & P(c) \\ 0 & 0 & 0 & 1.0 \end{pmatrix} \tag{4.2}$$

where, to the left of the matrix, for each of its rows, we show the corresponding states (in curly brackets). In this case, l=4, k=1 and N is of size 3×3. ■

Through this re-arrangement, we can use the following theorem [59]:

Theorem 4.1.1 Given a transition probability matrix Π of a homogeneous Markov chain Y_t in the form of Eq. (4.1), the probability for the time index n when the system first enters the set of absorbing states can be obtained from

$$P(Y_n \in A, Y_{n-1} \notin A, ..., Y_1 \notin A \mid \xi_{init}) = \xi^T N^{n-1}(I-N)\mathbf{1} \tag{4.3}$$

A denotes the set of absorbing states. $\mathbf{1}$ is simply a $(l-k) \times 1$ vector with all its elements equal to 1.0. ξ_{init} is the initial distribution on the states, i.e., it is a vector whose element i holds the probability that the PMC is in state i at the start. ξ consists of the $l-k$ elements of ξ_{init} corresponding to non-absorbing states.

In the theory of Markov chains, the current state of the chain is not always known and must be encoded in such a vector. For example, for the PMC of Figure 4.1b, we could have $\xi_{init}^{T} = (0.2\ 0.3\ 0.4\ 0.1)$, meaning that we are in state 0 with probability 20%, in state 1 with probability 30%, etc. However, in our case, at each point, the current state of $DFA_{\Sigma^{*}\cdot R}$ (and therefore of PMC_{R}^{m}) is known and therefore this vector would have 1.0 as the value for the element corresponding to the current state (and 0 elsewhere). ξ changes dynamically as the DFA/PMC moves among its various states and every state has its own ξ, denoted by ξ_q:

$$\xi_q(i) = \begin{cases} 1.0 & \text{if row } i \text{ of } N \text{ corresponds to state } q \\ 0 & \text{otherwise} \end{cases}$$

■ **Example 4.5** For the example of Figure 4.1b and for its transition matrix given by Equation (4.2), we have:

$$\xi_0{}^T = (1\ 0\ 0) \quad \xi_1{}^T = (0\ 1\ 0) \quad \xi_2{}^T = (0\ 0\ 1)$$

■

A slight variation of Equation 4.3 then gives the probability of the waiting-time variable:

$$P(W_R(q) = n) = \xi_q{}^T N^{n-1}(I-N)\mathbf{1}$$

4.2 Implementation

We implemented a forecasting system, Wayeb, based on Pattern Markov Chains. Algorithm 1 presents in pseudo-code the steps taken for recognition and forecasting. Wayeb reads a given pattern R in the form of a regular expression, transforms this expression into a NFA and subsequently, through standard determinization algorithms, the NFA is transformed into a *m-unambiguous* DFA (line 1 in Algorithm 1). For the task of event recognition, only this DFA is involved. At the arrival of each new event (line 7), the engine consults the transition function of the DFA and updates the current state of the DFA (line 8). Note that this function is simply a look-up-table, providing the next state, given the current state and the type of the new event. Hence, only a memory operation is required.

4.2.1 Learning the matrix of the PMC

To perform event forecasting, we need to create PMC_{R}^{m} and estimate its transition matrix. This is achieved by using the maximum-likelihood estimators for the transition probabilities of the matrix [92]. Let Π denote the transition matrix of a 1-order Markov chain, $\pi_{i,j}$ the transition probability from state i to state j and $n_{i,j}$ the number of transitions from state i to state j. Then, the maximum likelihood estimator for $\pi_{i,j}$ is given by:

$$\hat{\pi}_{i,j} = \frac{n_{i,j}}{\sum_{k \in Q} n_{i,k}} = \frac{n_{i,j}}{n_i} \tag{4.4}$$

where n_i denotes the number of visits to state i. Note also that we slightly abuse notation in the above formula, by using the symbol Q, which usually refers to the set of states of the DFA, to also denote the set of states of the PMC.

In order to obtain a realization of the sequence Y of the states that $DFA_{\Sigma^{*}\cdot R}$ visits and the observed values $\hat{\pi}_{i,j}^{obs}$ as estimates for the transition probabilities, we can use an initial

Algorithm 1: Wayeb

```
   Input: Stream $S$, pattern $R$, order $m$, maximum spread $ms$, forecasting threshold $\theta_{fc}$
   Output: For each event $e \in S$, a forecast $I = (start, end)$
1  $DFA_{\Sigma^* \cdot R}$ = BuildDFA($R$, $m$);
2  $PMC_R^m$ = WarmUp($S$, $DFA_{\Sigma^* \cdot R}$);
3  $F_{table}$ = BuildForecastsTable($PMC_R^m$, $\theta_{fc}$, $ms$);
4  $CurrentState$ = 0;
5  $RunningForecasts$ = $\varnothing$;
6  repeat
7      $e$ = RetrieveNextEvent($S$);
8      $CurrentState$ = UpdateDFA($DFA_{\Sigma^* \cdot R}$, $e$);
9      if $CurrenState$ not final then
10         $I = F_{table}(CurrentState)$;
11         $RunningForecasts = I \cup RunningForecasts$
12     else
13         UpdateStats($RunningForecasts$);
14         $RunningForecasts$ = $\varnothing$;
15     end
16 until $true$;
```

warm-up period during which a part of the stream is fed into the engine, the number of visits and transitions are counted and the transition probabilities are calculated, as per Equation (4.4) (line 2 in Algorithm 1).

4.2.2 Building forecasts

After estimating the transition matrix, PMC_R^m is used in order to compute the waiting-time distributions for each non-final state. Based on these waiting-time distributions, we build the forecasts associated with each state (line 3). A forecast produced by Wayeb is in the form of an interval $I = (start, end)$. The meaning of this interval is the following: at each point, the DFA is in a certain state. Given this state, we forecast that the DFA will have reached its final state (and therefore the pattern fully matched) at some future point between $start$ and end, with probability at least θ_{fc}. The calculation of this interval is done by using the waiting-time distribution that corresponds to each state and the threshold θ_{fc} is set beforehand by the user.

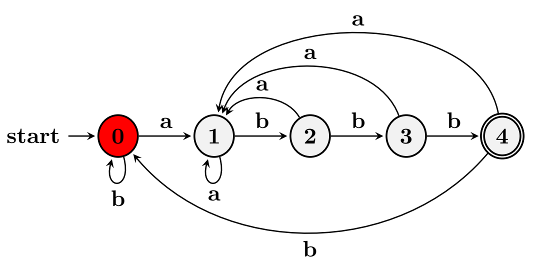
Each interval I that may be defined on the waiting-time distribution has an associated probability, given by:

$$P(I) = \sum_{n \in I} P(W_R(q) = n)$$

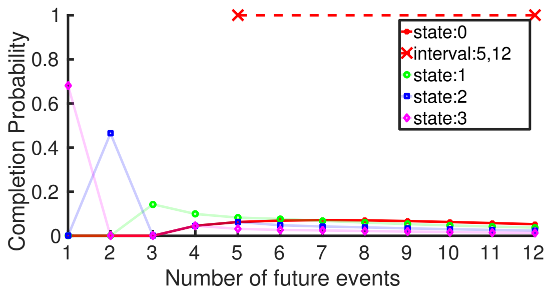
where we sum the probabilities of all points n that fall within I ($start \leq n \leq end$, where n is discrete). We define the set of intervals I_{fc} as:

$$I_{fc} = \{I : P(I) \geq \theta_{fc}\}$$

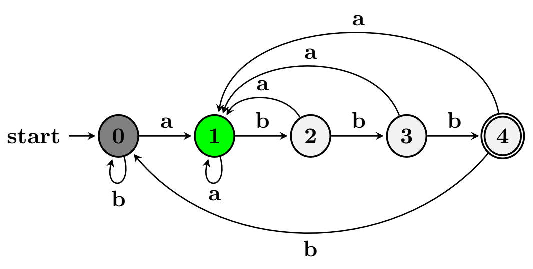
i.e., out of all possible intervals, I_{fc} contains those that have a probability above the user-defined threshold θ_{fc}. Any one of the intervals in I_{fc} may be provided as a forecast. However, a lengthy interval (e.g., the whole domain of the distribution has probability



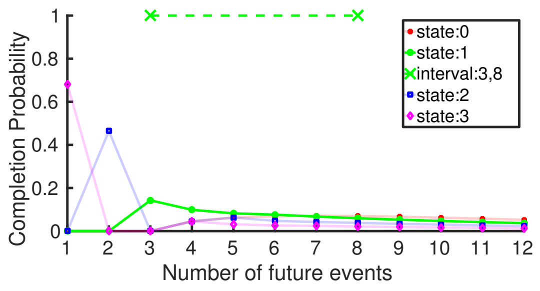
(a) DFA, state 0.



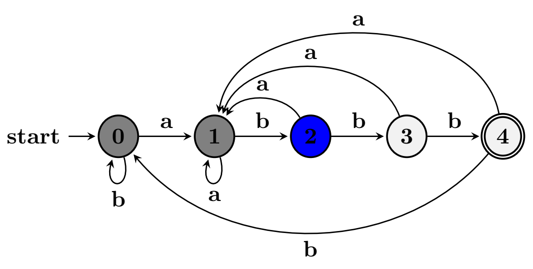
(b) Waiting-time distribution, state 0.



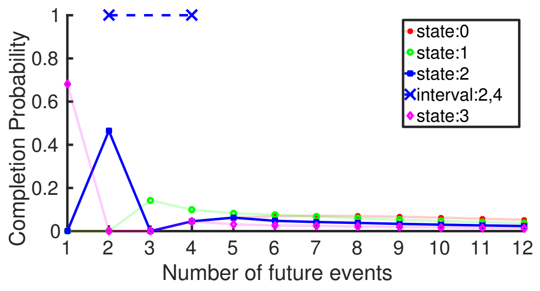
(c) DFA, state 1.



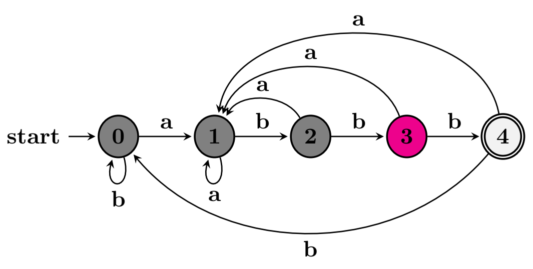
(d) Waiting-time distribution, state 1.



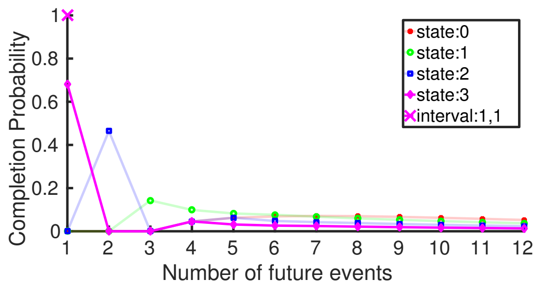
(e) DFA, state 2.



(f) Waiting-time distribution, state 2.



(g) DFA, state 3.



(h) Waiting-time distribution, state 3.

Figure 4.2: Example of how forecasts are produced. The pattern R is a sequential pattern $R = a \cdot b \cdot b \cdot b$ (one event of type a followed by three events of type b). $\Sigma = \{a, b\}$ (only two event types may be encountered, a and b) and $m = 1$. No maximum threshold for spread is set. $\theta_{fc} = 0.5$. For illustration purposes, the x axes stop at 12 future events.

100% and therefore always belongs to I_{fc}) is less informative than a small one. Therefore, out of all the intervals in I_{fc}, we wish to provide the one that has the smallest length. We define the spread of an interval as:

$$spread(I) = end - start$$

The forecast is therefore given as:

$$I_{best} = \arg\min_{I \in I_{fc}} spread(I) \tag{4.5}$$

If more than one interval with the same smallest spread exist, then we choose the one with the highest probability. From each waiting-time distribution, we extract the best interval, as defined by Equation (4.5), using a single-pass algorithm that scans the distribution of each state only once. We may additionally require that the spread of the forecast interval is no greater than a specified maximum threshold *ms* (see relevant input argument in line 3 of Algorithm 1). In this case, however, it might not be possible to find an interval that satisfies both constraints

$$P(I) \geq \theta_{fc} \wedge spread(I) \leq ms$$

and the algorithm will return an empty interval.

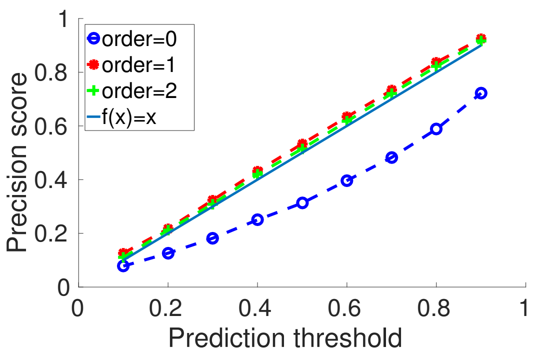
■ **Example 4.6** An example of how forecasts are produced is shown in Figure 4.2. The pattern R is a simple sequential pattern $R = a \cdot b \cdot b \cdot b$ (one event of type a followed by three events of type b). Also $\Sigma = \{a, b\}$ (only two event types may be encountered, a and b) and $m = 1$. Therefore, the distributions are calculated based on the conditional probabilities $P(a|a)$, $P(a|b)$, $P(b|a)$ and $P(b|b)$. No maximum threshold for the spread has been set in this example. As shown in Figure 4.2a, the DFA has 5 states (0-4) and state 4 is the final state. When no event has arrived (or only b events have arrived), the DFA is in its start state. The waiting-time distribution for this state is shown in Figure 4.2b as the red curve. The other distributions are shown as well, but they are greyed out, indicating that only the red curve is "activated" in this state. If the user has set $\theta_{fc} = 0.5$, then the best interval that Wayeb can produce is the one shown above the distributions (red, dashed line), and this is $I = (5, 12)$. Notice that, as expected, according to the red distribution, it is impossible that the pattern is fully matched within the next three events (it is in the start state and needs to see at least 4 events). If an a event arrives, the DFA moves to its next state, state 1 (Figure 4.2c), and now another distribution is "activated" (green curve, Figure 4.2d). The best interval is now $I = (3, 8)$ and has a smaller spread. The arrival of a b event activates the blue distribution (Figure 4.2f) and this time an even smaller interval is produced, $I = (2, 4)$. If a second b event arrives, the magenta distribution is activated. This distribution has a peak above 0.5 which is the value of the threshold θ_{fc} and this allows the engine to produce an interval with a single point $I = (1, 1)$. Essentially, Wayeb informs us that, with probability at least 50%, we will see a full match of the pattern in exactly 1 event from now. ■

Note that the calculation of the forecast intervals for each state needs to be performed only once, since for the same state it results always in the same interval being computed (assuming stationarity, as stated in Section 4.1). Therefore, the online forecasting system is again composed of a simple look-up-table (F_{table} in line 3 of Algorithm 1) and only memory operations are required.

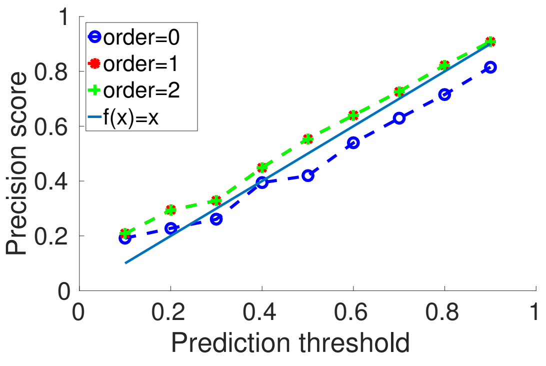
4.2.3 Performance and quality metrics

There are three metrics that we report in order to assess Wayeb's performance and the quality of its forecasts:

- $Precision = \frac{\#\ of\ correct\ forecasts}{\#\ of\ forecasts}$. At every new event arrival, the new state of the DFA is estimated (line 8 of Algorithm 1). If the new state is not a final state, a new



(a) $R = a \cdot b \cdot c$.



(b) $R = a \cdot (a+b)^* \cdot c$.

Figure 4.3: Precision scores for synthetic data, produced by a 1-order Markov process, with $\Sigma = \{a,b,c\}$.

forecast is retrieved from the look-up-table of forecasts (line 10). These forecasts are maintained in memory (line 11) until a full match is detected. Once a full match is detected, we can estimate which of the previously produced forecasts are satisfied, in the sense that the full match happened within the interval of a forecast (line 13). These are the correct forecasts. All forecasts are cleared from memory after a full match (line 14).

- $Spread = end - start$, as described in Section 4.2.2.
- $Distance = start - now$. This metric captures the distance between the time the forecast is made (*now*) and the earliest expected completion time of the pattern. Note that two intervals might have the same spread (e.g., $(2,2)$ and $(5,5)$ both have *Spread* equal to 0) but different distances (2 and 5, assuming $now = 0$).

Precision should be as high as possible. With respect to *Spread*, the intuition is that, the smaller it is, the more informative the interval. For example, in the extreme case where the interval is a single point, the engine can pinpoint the exact number of events that it will have to wait until a full match. On the other hand, the greater the *Distance*, the earlier a forecast is produced and therefore a wider margin for action is provided. Thus, "good" forecasts are those with high precision (ideally 1.0), low spread (ideally 0) and a distance that is as high as possible (ideal values depend on the pattern). These metrics may be calculated either as aggregates, gathering results from all states (in which case average values for *Spread* and *Distance* over all states are reported), or on a per-state basis, i.e., we can estimate the *Precision*, *Spread* and *Distance* of the forecasts produced only by a specific state of the DFA. We omit results for *Recall* (defined as percentage of detected events correctly predicted by at least one forecast), because *Recall* values are usually very high and not informative.

4.2.4 Validation tests with synthetic data

In this section we present the results from a series of tests on synthetic datasets. Each dataset is provided to Wayeb as a stream of events. Part of the stream (usually $\approx 20\%$) is used in order to learn the transition matrix Π. In each run, only a single pattern is tested and the order m, the forecasting threshold θ_{fc} and the maximum spread allowed *ms* are given by the user.

A set of tests was conducted with synthetically generated data for validation purposes. Streams were generated by a known Markov process and subsequently the engine was tested on these streams, for various patterns, forecast thresholds and orders. Figure 4.3 shows the aggregate (from all states) precision scores for two patterns, tested against a stream produced by a 1-order Markov process. The first pattern is the simple sequence $R = a \cdot b \cdot c$. The second pattern, $R = a \cdot (a+b)^* \cdot c$, is more complex and involves a *star closure* operation on the *union* of a and b, right after an a event and before a c event. For each pattern, three different values of the order m of the PMC were used (0, 1 and 2). The figures show how the engine behaves when the forecast threshold is increased and the order m of the PMC changes.

Note that the line $f(x) = x$ is also included in the figures, which acts as the baseline performance of the engine. If the Markov chain constructed for the pattern under test is indeed correct, then the precision score should lie above this line (or very close). As described in Section 4.2.2, each interval has a probability on the waiting-time distribution of at least $\theta_{fc} = x$. Therefore, if these waiting-time distributions are indeed correct, a percentage of at least θ_{fc} of the intervals will be satisfied. This also means that the actual precision score might be significantly higher than the threshold in cases where the waiting-time distributions have high peaks. For example, in Figure 4.2h the single-point interval produced has a probability of $\approx 70\%$, hence $\approx 70\%$ of forecasts from that state will be satisfied, which is significantly higher than the 50% forecasting threshold.

For both of the tested patterns, when $m = 0$ (blue curves) the precision scores are below the baseline performance, indicating that a PMC without memory is unable to produce satisfactory forecasts for a 1-order stream. When m is increased to match the order of the generating process ($m = 1$, red curves), the precision score does indeed lie above the baseline. A further increase in the value of m does not seem to affect the precision score (in Figure 4.3b, the red and green curves for $m = 1$ and $m = 2$ respectively coincide completely and only the latter is visible).

In some cases, even if we use an incorrect order m, the precision score may be above the baseline or even above the line of the correct order m. This may happen because incorrect models may produce "pessimistic" intervals, with high spread and therefore implicitly take a bigger "chunk" out of the correct distributions. In practice, however, the spread is constrained for informative forecasts, and thus models with incorrect order are insufficient.

4.3 Experimental results

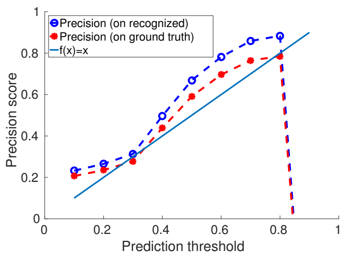
We present results from experiments on real-world datasets from credit card fraud management and maritime monitoring.

4.3.1 Credit card fraud management

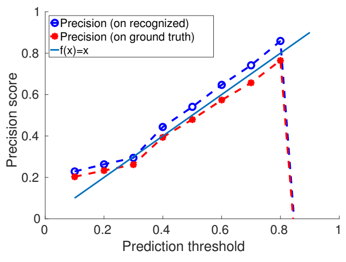
Unlike most academic and industrial work on fraud management, we performed an evaluation on a *real* dataset of credit card transactions, made available by Feedzai[1], our partner in the SPEEDD project[2]. Each event is a transaction accompanied by several arguments, such as the time of the transaction, the card ID, the amount of money spent, etc. There is also one boolean argument, indicating whether the transaction was labeled by a (human) analyst as being fraudulent or not. The original dataset is highly imbalanced. Only $\approx 0.2\%$ of the transactions are fraudulent. We created a summary of this original dataset, in which all fraudulent transactions were kept, but only some of the normal ones, so that the percentage

[1] https://feedzai.com/

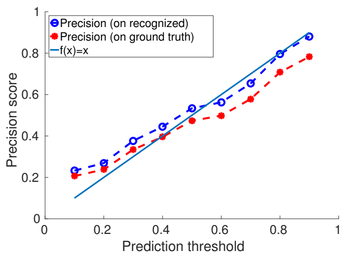
[2] http://speedd-project.eu/



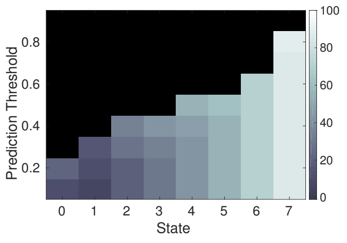
(a) Precision (all states), $m = 1$.



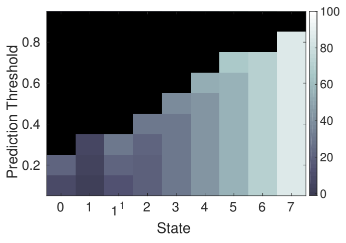
(b) Precision (all states), $m = 2$.



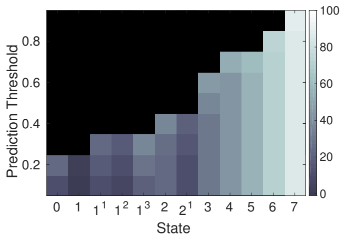
(c) Precision (all states), $m = 3$.



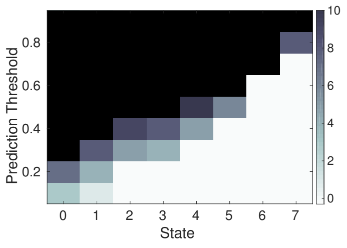
(d) Precision (per state), $m = 1$.



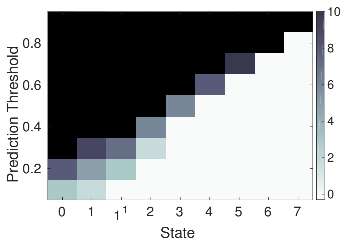
(e) Precision (per state), $m = 2$.



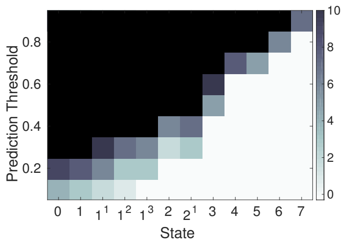
(f) Precision (per state), $m = 3$.



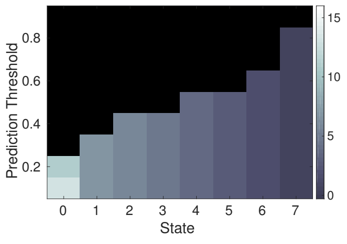
(g) Spread (per state), $m = 1$.



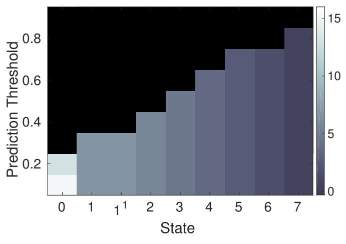
(h) Spread (per state), $m = 2$.



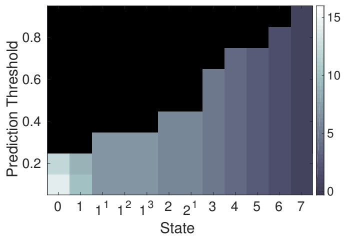
(i) Spread (per state), $m = 3$.



(j) Distance (per state), $m = 1$.



(k) Distance (per state), $m = 2$.



(l) Distance (per state), $m = 3$.

Figure 4.4: Results for the *IncreasingAmounts* pattern, for $m = 1$, $m = 2$ and $m = 3$, and for maximum spread $ms = 10$.

of fraudulent transactions rises to $\approx 30\%$. The total number of transactions in this summary dataset was ≈ 1.5 million.

In order to be able to detect fraudulent transactions, companies use domain expert knowledge and machine learning techniques, so that they can extract a set of patterns, indicative of fraud. For our experiments, we used a set of fraud patterns provided by Feedzai, our partner in the SPEEDD project. We also employed the partition-contiguity selection strategy, where the ID of a card is used as the partition attribute. Upon the arrival of a new transaction event, the ID is checked and the event is pushed to the PMC run that is responsible for this ID or a new run is created, in case this transaction is the first one for this card.

In Figure 4.4, the results for the pattern *IncreasingAmounts* are presented, for three different values of the order m (1, 2 and 3), where we have set the maximum allowed spread at the value of 10. This pattern detects 8 consecutive transactions of a card in which the amount of money in a transaction is higher than the amount in the immediately previous transaction (for the same card ID), i.e., it attempts to detect sequences of transactions with increasing trends in their amounts. Since such direct relational constraints are not currently supported by our engine, a pre-processing step was necessary. During this step, each transaction is flagged as either being *Normal* or as one having an *IncreasingAmount* with respect to the immediately previous. Therefore, the pattern provided to Wayeb starts with one *Normal* transaction, followed by 7 transactions flagged as *IncreasingAmount*.

Since, in this dataset, there is ground truth available by fraud analysts, indicating whether a transaction was fraudulent or not, besides measuring precision with respect to the events detected by the PMC, we can also measure precision with respect to those fraud instances that were both detected and were actually marked as fraudulent. The red curves in the precision figures correspond to precision scores as measured when ground truth is taken into account. Note, however, that the dataset annotation does not contain information about the fraud type. This means that, when we detect a match of the *IncreasingAmounts* pattern and the ground truth informs us that the involved transactions are indeed fraudulent, there is no way to determine whether they are considered as fraudulent due to a trend of increasing amounts or to some other pattern. As a result, the red curves could be "optimistic".

For all three values of the order m, Wayeb can maintain a precision score that lies above the $f(x)=x$ line (Figures 4.4a and 4.4b) or is very close to it (Figure 4.4c), i.e., the produced forecasts, compared against the recognized matches (blue curves), satisfy the threshold constraint. However, when $m=1$ or $m=2$ and $\theta_{fc}=0.9$, Wayeb cannot find intervals whose probability is at the same time above this threshold and whose spread is below 10, and fails to produce any forecasts (the sudden drop in the curves indicates forecast unavailability). By increasing the order to $m=3$ and taking more past events into account (Figure 4.4c), Wayeb can handle this high forecast threshold (we will come back to this issue at the end of this section). As compared against the ground truth (red curves), the precision scores are lower. This precision discrepancy between scores estimated against recognized matches and scores estimated against ground truth is due to the fact that the fraud pattern is imperfect, i.e., there are cases with 8 consecutive transactions with *IncreasingAmount* which do not actually constitute fraud. It is interesting to note, though, that the shape of the red curves closely follows that of the blue ones, indicating that, by using a more accurate pattern, we would indeed be able to achieve ground truth precision closer to that of the blue curves for all values of θ_{fc}.

The precision scores of Figures 4.4a, 4.4b and 4.4c are calculated by combining the forecasts produced by all states of the PMC. In order to better understand Wayeb's behavior, a look at the behavior of individual states could be more useful. Figures 4.4d – 4.4l depict image plots for various metrics against both the forecast threshold and the state of the PMC. The metrics shown are those of precision (on the recognized matches), spread and distance. We omit the plots for ground truth precision because they have the same shape as those for precision on recognized matches, but with lower values. In each such image plot the y axis corresponds to the various values of θ_{fc}. The x axis corresponds to the states of the PMC. Each state has a unique integer identifier, starting from 0 (the start state). We group together states that are duplicates of each other, in cases where some states are ambiguous. For example, in Figure 4.4f, states 1^1, 1^2 and 1^3 are all duplicates of state 1. In this way, the x axis shows how advanced we are in the recognition process, when moving from one cluster of duplicates to the next. The black areas in these plots are "dead zones", meaning that, for the corresponding combinations of θ_{fc} and state, Wayeb fails to produce forecasts (i.e., it cannot guarantee, according to the learned transition probabilities, that the forecast intervals will have at least θ_{fc} probability of being satisfied). On the contrary, areas with light colors are "optimal", in the sense that they have high precision, low spread (the colorbar is inverted in the spread plots) and high distance in their respective plots.

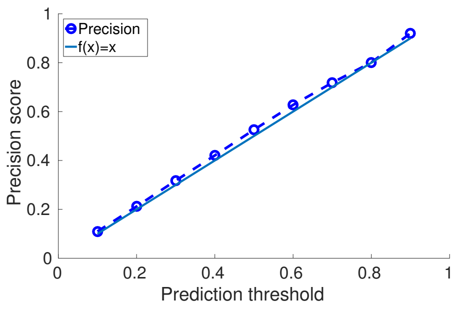
The precision plots (4.4d, 4.4e, 4.4f) show that the more advanced states of the PMC enter into such dead zones at higher forecast thresholds. Figures 4.4g, 4.4h and 4.4i show the spread of the forecast intervals. Two clearly demarcated zones emerge. One is the usual dead zone (black, top left). The other one (white, bottom right) corresponds to forecasts whose spread is 0, i.e., single point forecasts. A common behavior for all states is that, as higher values of θ_{fc} are set to the engine, the spread increases, i.e., each state attempts to satisfy the constraint of the forecast threshold by taking a longer interval from the waiting-time distribution. On the other hand, those states that are more advanced can maintain small spread values for a wider margin of θ_{fc} values. For example, in Figure 4.4g, state 2 maintains a spread of 0 until θ_{fc}=0.2 whereas state 5 hits this limit at around θ_{fc}=0.5. Figures 4.4j, 4.4k and 4.4l show image plots for the distance metric. As can be seen, those regions that have high precision scores and low spread, also tend to have low distance. Therefore, there is a trade-off between these three metrics. For the case of m=1, good forecasts might be considered those of state 5, which can maintain a small spread until θ_{fc}=0.5 and whose temporal distance is ≈ 3. By increasing m, one can get good forecasts that are more "satisfactory", at the cost of an increased size for the PMC (a discussion about this cost will be presented shortly). For example, when m=2, as in Figure 4.4h, state 5 can produce single point forecasts for higher values of θ_{fc} (for 0.6 too).

As a final comment, we note that increasing the value of m does not necessarily imply higher precision scores. In fact, as shown in Figures 4.4a, 4.4b, 4.4c, the precision score might even decrease. This behavior is due to the fact that, in general, smaller values of m tend to produce more "pessimistic" intervals, with higher spread. For example, for θ_{fc}=0.8, the precision score for m=1 (Figure 4.4a) is in fact higher than when m=2 (Figure 4.4b). In Figure 4.4g, we can see that, for m=1, the intervals of state 7 when θ_{fc}=0.8 (the only state producing forecasts for this value of θ_{fc}) have a high spread whereas the same state, when m=2, produces intervals with low spread (Figure 4.4h). Since "pessimistic", high-spread intervals take a bigger "chunk" out of a distribution, their precision scores end up being also higher. By increasing m, Wayeb can approximate the real waiting-time distributions more closely and therefore produce forecasts with lower spread that are closer to the specified threshold. Therefore, the accuracy curve (blue, dashed curve) starts to coincide with the $f(x)$=x line.

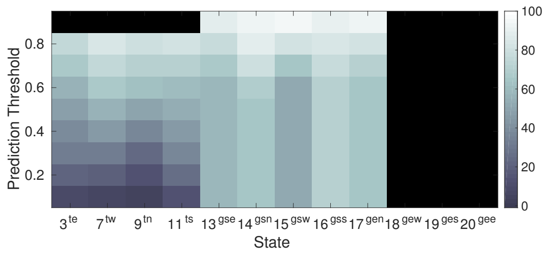
4.3.2 Maritime monitoring

Another real-world dataset against which Wayeb was tested came from the field of maritime monitoring. When sailing at sea, (most) vessels emit messages relaying information about their position, heading, speed, etc.: the so-called AIS (automatic identification system) messages. AIS messages may be processed in order to produce a compressed trajectory, consisting of critical points, i.e., important points that are only a summary of the initial trajectory, but allow for an accurate reconstruction [113]. The critical points of interest for our experiments are the following:

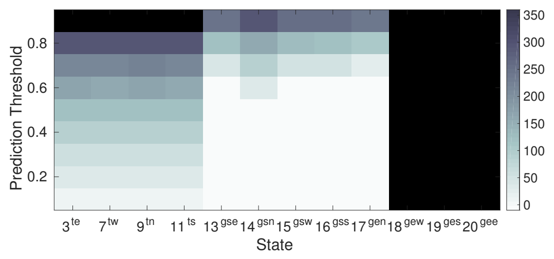
- *Turn*: when a vessel executes a turn.
- *GapStart*: when a vessel turns off its AIS equipment and stops transmitting its position.
- *GapEnd*: when a vessel turns on its AIS equipment back again (a *GapStart* must



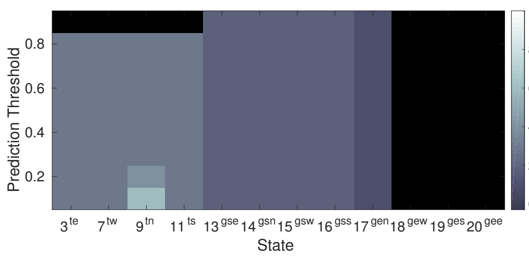
(a) Precision (all states).



(b) Precision (all states).



(c) Precision (all states).



(d) Precision (per state).

Figure 4.5: Results for the pattern $Turn \cdot GapStart \cdot GapEnd \cdot Turn$ with $m = 1$.

have preceded).

We used a dataset consisting of a stream of such critical points from $\approx$ 6.500 vessels, covering a 3 month period and spanning the Greek seas. Each critical point was enriched with information about whether it is headed towards the northern, eastern, southern or western direction. For example, each *Turn* event was converted to one of *TurnNorth*, *TurnEast*, *TurnSouth* or *TurnWest* events. We show results from a single vessel, with $\approx$ 50.000 events.

Figure 4.5 shows results for the pattern

$$Turn \cdot GapStart \cdot GapEnd \cdot Turn \tag{4.6}$$

where *Turn* is shorthand notation for

$$(TurnNorth + TurnEast + TurnSouth + TurnWest)$$

with $+$ denoting the *OR* operator. Similarly for *GapStart* and *GapEnd*. With this pattern, we would like to detect a sequence of movements in which a vessel first turns (regardless of heading), then turns off its AIS equipment and subsequently re-appears by turning again. Communication gaps are important for maritime analysts because they often indicate an intention of hiding (e.g., in cases of illegal fishing in a protected area). The aggregate precision score (Figure 4.5a) is very close to the baseline performance. A look at the per-state plots reveals something interesting (Figures 4.5b, 4.5c, 4.5d). Note that, in order to avoid cluttering, we have removed duplicate states from the per-state plots. In addition, the superscript of each state in the x axis shows the last event seen when in that state. For example, the superscript te corresponds to *TurnEast*, tw to *TurnWest*, tn to *TurnNorth* and ts to *TurnSouth* (states 3, 7, 9 and 11 respectively). Similarly for *GapStart* for which superscripts start with gs (states 13–16) and for *GapEnd* (ge and states 17–20). These per-state plots show that there is a distinct “cluster” of states (13–17) which exhibit high precision scores for all values of θ_{fc} (Figure 4.5b) and small spread for most values of θ_{fc}

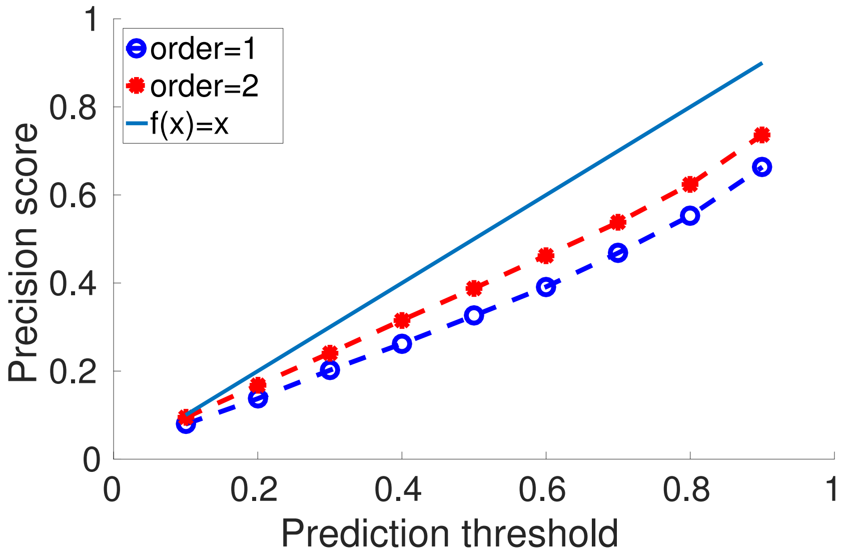


Figure 4.6: Results for the pattern $TurnNorth \cdot (TurnNorth + TurnEast)^* \cdot TurnSouth$.

(Figure 4.5c). Therefore, these states constitute what might be called “milestones” and a PMC can help in uncovering them. By closer inspection, it is revealed that states 13–16 are visited after the PMC has seen one of the *GapStart* events (we remind that *GapStart* is a disjunction of the four directional sub-cases). Moreover, *GapEnd* events are very likely to appear in the input stream right after a *GapStart* event, as expected, since during a communication gap (delimited by a *GapStart* and a *GapEnd*), a vessel does not emit any messages. State 17, which also has a similar behavior, is visited after a *GapEndNorth* event. Its high precision scores are due to the fact that, after a *GapEnd* event, a *Turn* event is very likely to appear. It differs from states 13–16 in its distance, as shown in Figure 4.5d, which is 1, whereas, for states 13–16, the distance is 2. On the other hand, states 18–20, which correspond to the other 3 *GapEnd* events, fail to produce any forecasts. The reason is that there are no such *GapEnd* events in the stream, i.e., whenever this vessel starts transmitting again after a *Gap*, it is always headed towards the northern direction.

Figure 4.6 shows results for the pattern

$$TurnNorth \cdot (TurnNorth + TurnEast)^* \cdot TurnSouth$$

This pattern is more complex since it involves a *star closure* operation on a nested *union* operation. It attempts to detect a rightward reverse of heading, in which a vessel is initially heading towards the north and subsequently starts a right turn until it ends up heading towards the south. Such patterns can be useful in detecting maneuvers of fishing vessels.

Figure 4.6 shows that a model with m=1 is unable to approximate well-enough the correct waiting-time distribution. Increasing the order to m=2 improves the precision score, but it still remains under the baseline performance. One could attempt to further increase the value of m, but this would substantially increase the cost of building the PMC. For $m = 1$, the generated PMC has ≈ 30 states. For $m = 2$, this number rises to ≈ 600 and the cost of creating an unambiguous DFA and then its corresponding PMC rises exponentially. When stationarity is assumed (as in our case) and the model does not need to be updated online, an expensive model can be tolerated.

4.3.3 Commentary on throughput

So far, we have focused on precision and quality metrics. For online event forecasting, throughput (defined as $\frac{\#\ of\ events\ consumed}{total\ execution\ time}$, where *execution time* refers to the **repeat** loop in Algorithm 1) is another important metric. We omit presenting detailed results about throughput, since Wayeb exhibits a steady behavior. For the maritime use case and the more complex heading reversal pattern, throughput is $\approx 1.2 \times 10^6$ $events/sec$ and remains steady for both m=1 and m=2, whereas the event rate of the input stream is much lower. The experiments for the maritime use case were run on a 64-bit Debian machine, with Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processors, and 16GB of memory. This high throughput number is due to the fact that the online operations of Wayeb consist mostly of memory operations (see Section 4.2). Even when the size of the PMC grows from ≈ 30 to ≈ 600, there is minimal overhead in accessing and maintaining the larger look-up-tables of the latter PMC. This independence from m also holds when multiple runs are employed, as in the credit card fraud use case (for the $partition - contiguity$ selection strategy). Even in this case, only a single PMC is created (therefore, only one table for the DFA and one for the forecasts) and the different runs simply consult this PMC through a reference to it. Throughput for the *IncreasingAmounts* fraud pattern is $\approx 1.2 \times 10^5$ $events/sec$ (in total, 3 different patterns were tested), whereas the event rate at peak times reaches up to ≈ 1000 $events/sec$. Due to privacy reasons, experiments on the fraud dataset were run in Feedzai's premises and thus on different hardware: a 64-bit Ubuntu machine, with Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processors and 32GB of memory. The multiple runs that need to be created, accessed and maintained with this dataset (on the contrary, for the maritime use case, only a single run is created) incur a significant increase in the execution time.

4.4 Summary

In this chapter we presented Wayeb, a system that can produce online forecasts of event patterns. This system is not restricted to sequential patterns, but can handle patterns defined as regular expressions. It is also probabilistic and its forecasts can have guaranteed precision scores, if the input stream is generated by a Markov process. We have shown that it can provide useful forecasts even in real-world scenarios in which we do not know beforehand the statistical properties of the input stream. Moreover, the trade-off between precision score and the quality of the produced forecasts has been explored. Wayeb can also be used to uncover interesting probabilistic dependencies among the events involved in a pattern (pattern "milestones"), which can be informative in themselves or could possibly be used for optimization purposes in algorithms based on frequency statistics [81].

5. Wayeb: a tool for complex event forecasting

In this chapter we extend Wayeb so that it can employ symbolic automata. Patterns are initially defined as symbolic regular expressions. They are then converted to symbolic automata that can consume event streams and detect pattern occurrences. Symbolic automata are also converted to Markov chains so that a probabilistic model of an automaton's behavior may be learnt. Based on this model, we can derive forecasts after every new event arrival about when a symbolic automaton is expected to reach a final state, i.e., about when a pattern is expected to occur.

5.1 Forecasting with symbolic automata

One limitation of the method presented in the previous chapter is that it requires a finite alphabet, i.e., the input stream may be composed only of a finite set of symbols; on the other hand, events in a stream are in the form of tuples and look more like data words. Their attributes might be real–valued, taking values from an infinite set, and thus cannot be directly handled by DFA. In order to overcome this limitation, we have extended our method so that symbolic automata are employed [44, 134]. Symbolic automata, instead of having symbols from a finite set on their transitions, are equipped with predicates from a Boolean algebra, acting as transition guards. Such predicates can reference any event attribute, thus allowing for more expressive patterns.

We now present a formal definition of symbolic automata (for details, see [44]).

Definition 5.1.1 — Symbolic Automaton. A symbolic finite automaton (*SFA*) is a tuple $M = (\mathscr{A}, Q, q^0, F, \Delta)$, where $\mathscr{A}$ is an effective Boolean algebra, Q is a finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\Delta \subseteq Q \times \Psi_{\mathscr{A}} \times Q$ is a finite set of transitions, with $\Psi_{\mathscr{A}}$ being s set of predicates closed under the Boolean connectives.

Most other definitions from classical automata carry over symbolic automata. Importantly, *SFA* are determinizable [44]. The definition for deterministic *SFA* is also similar to that for classical automata, with the important difference that it is not enough for all transitions from a state to have different predicates. We require that predicates on transitions from the same state are mutually exclusive, i.e., at most one may evaluate to TRUE (see again [44]). The determinization process for *SFA* is similar to that for classical automata and is based on the construction of the power–set of the states of the non–deterministic automaton. Due to this issue of different predicates possibly both evaluating to TRUE for the same event/tuple, we first need to create the *minterms* of the predicates of a *SFA*, i.e., the set of maximal satisfiable Boolean combinations of such predicates. When constructing a deterministic *SFA*, these minterms are used as guards on the transitions, since they are mutually exclusive.

This result about *SFA* being determinizable allows us to use same technique of converting a deterministic automaton to an m–unambiguous automaton and then to a Markov chain. First, note that the set of minterms constructed from the predicates appearing in a *SFA* M, denoted by $Minterms(Predicates(M))$, induces a finite set of equivalence classes on the (possibly infinite) set of domain elements of M [44]. For example, if $Predicates(M) = \{\psi_1, \psi_2\}$, then $Minterms(Predicates(M)) = \{\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \neg\psi_2\}$ and we can map each domain element, which, in our case, is an event/tuple, to exactly one of these 4 minterms: the one that evaluates to TRUE when applied to the element. Similarly, the set of minterms induces a set of equivalence classes on the set strings (event streams in our case). For example, if $S{=}t_1, \cdots, t_n$ is an event stream, then it could be mapped to $S^{'}{=}a, \cdots, b$, with a corresponding to $\psi_1 \wedge \neg\psi_2$ if $\psi_1(t_1) \wedge \neg\psi_2(t_1) = $ TRUE, b to $\psi_1 \wedge \psi_2$, etc. We say that $S^{'}$ is the stream induced by applying $Minterms(Predicates(M))$ on the original stream S. We first give a definition for an an m–unambiguous deterministic *SFA*, by modifying the relevant definition for classical automata [107]:

Definition 5.1.2 — m–unambiguous deterministic *SFA*. A deterministic *SFA* $(\mathscr{A}, Q, q^0, F, \Delta)$ is m–ambiguous if there exist $q \in Q$ and $a, b \in T^m$ such that $a \neq b$ and $\delta(q,a) = \delta(q,b)$. A deterministic *SFA* which is not m–ambiguous is m–unambiguous.

In other words, upon reaching a state q of an m–unambiguous *SFA*, we know which last m minterms evaluated to TRUE, i.e., we know the last m symbols of the induced stream $S^{'}$. The following proposition then follows:

Proposition 5.1.1 Let M be a deterministic m–unambiguous *SFA*, $S = t_1, \cdots, t_n$ an event stream and $S^{'} = X_1, \cdots, X_n$ the stream induced by applying $T = Minterms(Predicates(M))$ on S. Assume that $S^{'}$ is a m–order Markov process, i.e., $P(X_i = a \mid X_1, \cdots, X_{i-1}) = P(X_i = a \mid X_{i-m}, \cdots, X_{i-1})$. Then the sequence $Y = Y_m, \cdots, Y_n$ defined by $Y_0 = q^0$ and $Y_i = \delta(Y_{i-1}, t_i)$ (i.e., the sequence of states visited by M) is a 1–order Markov chain whose transition matrix is given by:

$$\Pi(p,q) = \begin{cases} P(X_{m+1} = a \mid X_1, \cdots, X_m = \phi(\delta^{-m}(p))) & \text{if } \delta(p,a) = q \\ 0 & \text{if } q \notin \delta(p,T) \end{cases}$$

where $\delta^{-m}(q) = \{a \in T^m \mid \exists p \in Q : \delta(p,a) = q\}$ is the set of concatenated labels of length m that can lead to q and, by definition, is a singleton for m–unambiguous *SFA*.

Proof. This result holds for classical m–unambiguous deterministic automata [107]. For the symbolic case, note that, from an algebraic point of view, the set $T = Minterms(Predicates(M))$ may be treated as a generator of the monoid T^*, with concatenation as the operation. If the cardinality of T is k, then we can always find a set $\Sigma = \{a_1, \cdots, a_k\}$ of k distinct symbols and then a morphism (in fact, an isomorphism) $\phi : T^* \rightarrow \Sigma^*$ that maps each minterm to exactly one, unique a_i. A classical deterministic automaton N can then be constructed by relabelling the *SFA* M under ϕ, i.e., by copying/renaming the states and transitions of the original *SFA* M and by replacing the label of each transition of M by the image of this label under ϕ. Then, the behavior of N (the language it accepts) is the image under ϕ of the behavior of M [122]. Based on these observations, we can see that the sequence of states visited by an m–unambiguous deterministic *SFA* is indeed a 1–order Markov chain, as a direct consequence of the fact that a deterministic *SFA* has the same behavior (up to isomorphism) to that of a classical deterministic automaton, constructed through relabelling. ■

Therefore, after constructing a deterministic *SFA* we can use it to construct a PMC and learn its transition matrix. The conditional probabilities in this case are essentially probabilities of seeing an event that will satisfy a predicate, given that the previous event(s) have satisfied the same or other predicates. For example, if $m = 1$ and $Predicates(M) = \{\psi_1, \psi_2\}$, one such conditional probability would be $P(\psi_1(t_{i+1}) \wedge \psi_2(t_{i+1}) = \textsf{TRUE} \mid \neg\psi_1(t_i) \wedge \neg\psi_2(t_i) = \textsf{TRUE})$, i.e., the probability of seeing an event that will satisfy both ψ_1 and ψ_2 given that the current, last seen event satisfies neither of these predicates. As with classical automata, we can then provide again forecasts based on the waiting–time distributions of the PMC derived from a *SFA*.

5.2 Experimental results

Wayeb is a Complex Event Forecasting engine based on symbolic automata, written in the Scala programming language. It was tested against two real–world datasets coming from the field of maritime monitoring. When sailing at sea, (most) vessels emit messages relaying information about their position, heading, speed, etc.: the so-called AIS (automatic identification system) messages. Such a stream of AIS messages can then be used in order to detect interesting patterns in the behavior of vessels [113]. Two AIS datasets were used, made available in the datAcron project[1]: the first contains AIS kinematic messages from vessels sailing in the Atlantic Ocean around the port of Brest, France, and span a period from 1 October 2015 to 31 March 2016 [119]; the second was provided by IMISG[2] and contains AIS kinematic messages from most of Europe (the entire Mediterranean Sea and parts of the Atlantic Ocean and the Baltic Sea), spanning an one–month period from 1 January 2016 to 31 January 2016. AIS messages can be noisy, redundant and typically arrive at unspecified time intervals. We first processed our datasets in order to produce clean and compressed trajectories, consisting of critical points, i.e., important points that are a summary of the initial trajectory, but allow for an accurate reconstruction [113]. Subsequently, we sampled the compressed trajectories by interpolating between critical points in order to get trajectories where each point has a temporal distance of one minute from its previous point. After excluding points with null speed (in order to remove stopped vessels), the final streams consist of ≈ 1.3 million points for the Brest dataset and ≈ 2.4 million points for the IMISG dataset. The experiments were run on a machine with Intel Core i7-4770 CPU @ 3.40GHz processors and 16 GB of memory.

[1]http://datacron-project.eu/

[2]https://imisglobal.com/

We have chosen to demonstrate our forecasting engine on two important patterns. The first concerns a movement pattern in which vessels approach a port and the goal is to forecast when a vessel will enter the port. This is a key target of several vessel tracking software platforms, as indicated by the 2018 challenge of the International Conference on Distributed and Event-Based Systems [69]. The second pattern concerns a fishing maneuver inside a fishing area. Forecasting when vessels are about to start fishing could be important in order to manage the pressure exerted on fishing areas [78].

The *approaching* pattern may be defined as follows:

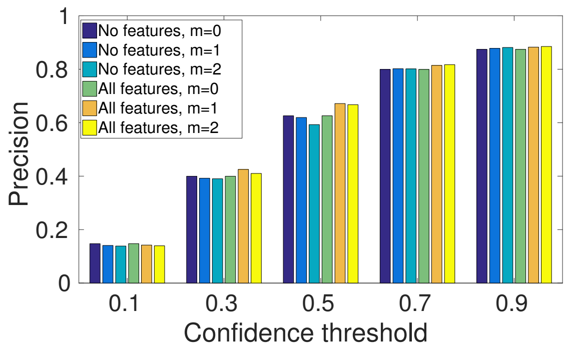
$$\begin{aligned}R_1 :=\ & x \cdot y^+ \cdot z \text{ WHERE}\\ & \mathit{Distance}(x, \mathit{PortCoords}, \mathit{7.0}, \mathit{10.0}) \text{ AND } \mathit{Distance}(y, \mathit{PortCoords}, \mathit{5.0}, \mathit{7.0}) \text{ AND}\\ & \mathit{WithinCircle}(z, \mathit{PortCoords}, \mathit{5.0}) \text{ PARTITION BY } \mathit{vesselId}\end{aligned} \tag{5.1}$$

Concatenation is denoted by $\cdot$ and $^+$ stands for Kleene+. We want to start detecting a vessel's movement whenever a vessel is between 7 and 10 km away from a specific port, then it approaches the port, with its distance from it falling to the range of 5 to 7 km, stays in that range for 1 or more messages, and finally enters the port, defined as being inside a circle with a radius of 5 km around the port. We have chosen the above syntax in order to make clear what the regular part of a pattern is (before the WHERE keyword) and what its logical part is (after WHERE), but predicates can be placed directly in the regular expression. Note also that the first argument of a predicate is the event upon which it is to be applied, but we also allow for other arguments to be passed as constants. The *partition contiguity* selection strategy is employed [143], i.e., for each new vessel appearing in the stream, a new automaton run is created, being responsible for this vessel. This is just a special case of parametric trace slicing typically used in runtime verification tools [33]. If we use this pattern to build a PMC, the transition probabilities will involve the three predicates that appear in it. However, it is reasonable to assume that other features of a vessel's kinematic behavior could affect the accuracy of forecasts, e.g., its speed. For this reason, we have also added a mechanism to our module that can incorporate in the PMC some extra features, declared by the user, but not present in the pattern itself. The extra features we decided to add concern the vessel's speed and its heading: $\mathit{SpeedBetween}(x, \mathit{0}, \mathit{10.0})$, $\mathit{SpeedBetween}(x, \mathit{10.0}, \mathit{20.0})$, $\mathit{SpeedBetween}(x, \mathit{20.0}, \mathit{30.0})$ and $\mathit{HeadingTowards}(x, \mathit{PortCoords})$. The first three try to use the speed level of a vessel, whereas the last one uses the vessel's heading and checks whether it is headed towards the port. Our experiments were conducted using the main Brest port as the port of reference.

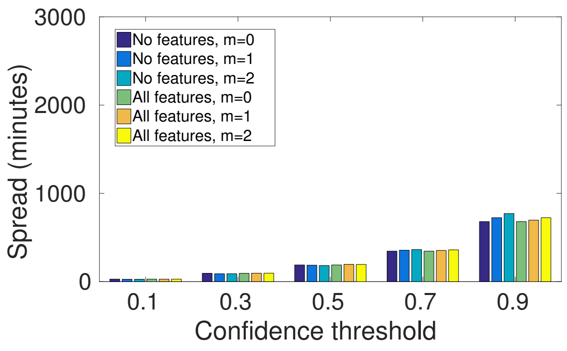
The *fishing* pattern may be defined as follows:

$$\begin{aligned}R_2 :=\ & x \cdot y^* \cdot z \text{ WHERE}\\ & (\mathit{IsFishingVessel}(x) \wedge \neg \mathit{InArea}(x, \mathit{FishingArea})) \text{ AND}\\ & (\mathit{InArea}(y, \mathit{FishingArea}) \wedge \mathit{SpeedBetween}(y, \mathit{9.0}, \mathit{20.0})) \text{ AND}\\ & (\mathit{InArea}(z, \mathit{FishingArea}) \wedge \mathit{SpeedBetween}(z, \mathit{1.0}, \mathit{9.0}))\\ & \text{PARTITION BY } \mathit{vesselId}\end{aligned} \tag{5.2}$$

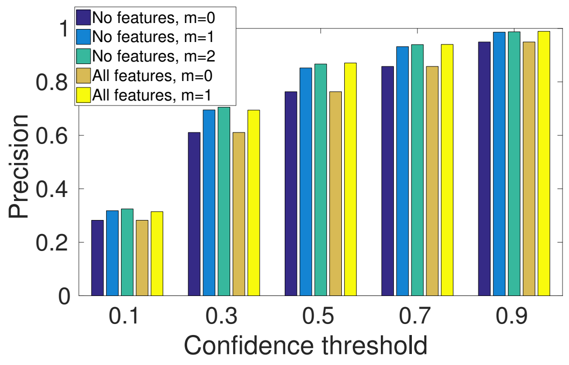
This definition attempts to capture a movement of a fishing vessel, in which it is initially outside a specific fishing area, then enters the area with a traveling speed (between 9 to 20 knots), remains there for zero or more messages, and finally starts moving with a fishing speed (between 1 to 9 knots) while still in the same area. We also used the same



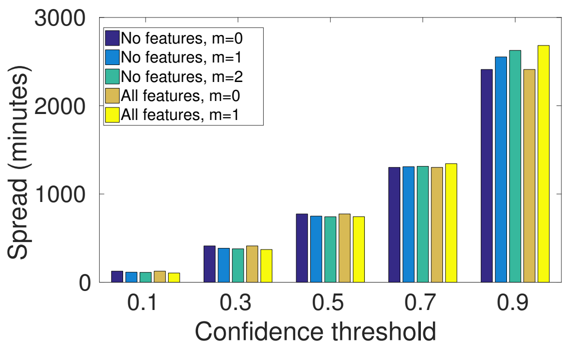
(a) Precision (approaching pattern).



(b) Spread (approaching pattern).



(c) Precision (fishing pattern).



(d) Spread (fishing pattern).

Figure 5.1: Results for approaching the Brest port and for fishing.

extra features as in Pattern 5.1, with the difference that the *Heading* feature now concerns the area. Note that by partitioning by *vesselId* we don't have to add the *IsFishingVessel* predicate for y and z.

Figure 5.1 shows *precision* and *spread* results on the Brest dataset for different values of the forecasting confidence threshold θ_{fc} and of the assumed order m. Precision is defined as the percentage of forecasts that were correct, i.e., whose interval includes the timestamp of the complex event's actual occurrence. Spread is defined as the length of a forecast interval. The smaller the spread, the more focused a forecast is and thus more valuable. Good results are therefore considered those with high precision scores (ideally 1.0) and low spread scores (ideally 0, i.e., single point forecasts). For each confidence threshold, each bar represents a different variation of the pattern. We vary the order m of the PMC in order to investigate the possible gains of looking deeper into the past. There are also variations where no extra features are used (i.e., only the predicates of the original pattern are included in the PMC) and where all the extra features are included. Our engine can always achieve precision scores that are above the confidence threshold. However, note that, as the confidence threshold increases, the spread also tends to increase. This happens because the engine tries to satisfy the θ_{fc} constraint by expanding the forecast intervals. Looking back at the example of Figure 4.2d, if, instead of $\theta_{fc} = 50\%$, we set $\theta_{fc} = 70\%$, then the green interval shown at the top would have to be expanded to the right so that its probability exceeds the new increased threshold (expanding to the left would be pointless since only points with zero probability exist in this region). It is interesting to also note that, when we include the extra features, the spread is lower for $\theta_{fc} = 0.9$, indicating that these features help in producing more focused forecasts. Although a similar pattern can be observed for the *fishing* pattern as well, a more careful examination reveals that this pattern

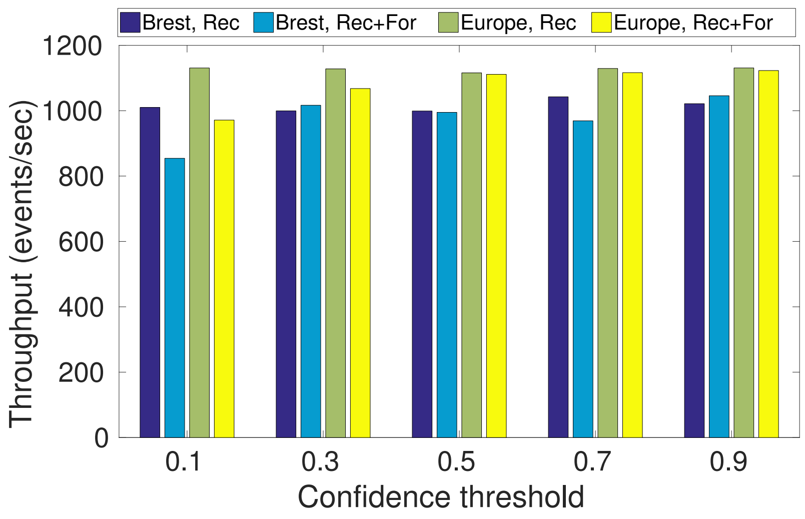


Figure 5.2: Throughput results. “Rec” denotes recongition only, with forecasting disabled. “Rec+For” denotes that both recognition and forecasting are enabled.

is more challenging. The precision scores are higher, but the spread is also significantly higher for all values of θ_{fc}. This means that accurately pinpointing when a fishing pattern will be detected is more difficult. This result is also expected, since fishing maneuvers (often occurring in the open sea) exhibit a greater degree of variability, whereas movement patterns while approaching a port are more or less straightforward. The trade-off between precision and spread is thus always present, but its exact nature also depends on both the pattern itself and the included features. We also see that increasing m does not seem to have a significant impact, at least, with the predicates chosen here. Note that this is not always the case, since m can indeed play a significant role in other domains and/or patterns [6]. As a general comment, Figure 5.1 can help a user determine a satisfactory set of parameter values. For example, for the approaching pattern, a user could choose to set $\theta_{fc} = 50\%$ and $m = 0$, which gives a high enough precision with relatively low spread and avoids the cost of disambiguation that accompanies any higher values of m.

In order to assess Wayeb's throughput, we run another series of experiments. We tested the *approaching* pattern on both the Brest and the Europe datasets. For the Brest dataset, all of the 222 ports of Brittany were included, each with its own *SFA* and PMC. For the Europe dataset, 222 European ports were randomly selected. We tested for throughput both when forecasting is performed and when it is disabled, in which case only recognition is performed. The results are shown in Figure 5.2. As expected, throughput is lower when forecasting is enabled. However, the overhead is not significant. We also see that throughput is higher for the Europe dataset. It is possible to achieve higher throughput because the event rate of the input stream is also higher. The throughput is higher in this case, despite the fact that it contains almost ten times as many vessels and has a higher incoming event rate, indicating that Wayeb can scale well as the number of monitored objects increases.

5.3 Summary

Wayeb is one of the few CER systems capable of the form of forecasting presented here. The advantage of our method is that it allows for a deeper investigation of the past, if this is needed, by being able to handle higher order Markov processes. By using symbolic automata, our computational model also has nice compositional properties (a feature generally lacking in CER) and can accommodate any regular expression, without being restricted to sequential patterns. Having predicates on the transitions also allows for an easy incorporation of Boolean expressions and of background knowledge. For example, note that the $IsFishingVessel(x)$ predicate in Pattern (5.2) is evaluated not by using information from the streaming AIS messages (which may not always contain correct information about a vessel's type) but by using the relevant background knowledge of the list of fishing vessels.

6. Forecasting with suffix trees: theory

In order to avoid the combinatorial explosion on the number of states of a PMC, as described in the previous chapters, we show in this chapter how we can employ variable–order Markov models [28]. We present a formal framework for CEF, along with an implementation and extensive experimental results on real and synthetic data from diverse application domains. Our framework allows a user to define a pattern for a complex event, e.g., a pattern for fraudulent credit card transactions or for two moving objects moving in close proximity and towards each other. 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 the formalism of symbolic automata [44] to encode a pattern and that of prediction suffix trees [120, 121] to learn a probabilistic model for the pattern. We formally show how symbolic automata can be combined with prediction suffix trees to perform CEF. Prediction suffix trees fall under the class of the so-called variable-order Markov models, i.e., 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.

Our empirical analysis shows the advantage of being able to use high-order models over related non-Markov methods for CEF and methods based on low-order Markov models (or Hidden Markov Models). The price we have to pay for this increased accuracy is a decrease in throughput, which still however remains high (typically tens of thousands of events per second). The training time is also increased, but still remains within the same order of magnitude. This fact allows us to be confident that training could also be performed online.

Our contributions may be summarized as follows:

- We present a CEF framework that is both formal and easy to use. It is often the case that CER frameworks lack clear semantics, which in turn leads to confusion about how patterns should be written and which operators are allowed [64]. This problem is exacerbated in CEF, where a formalism for defining the patterns to be forecast may be lacking completely. Our framework is formal, compositional and as easy to use as writing regular expressions. The only basic requirement is that the user declaratively define a pattern and provide a training dataset.

- Our framework can uncover deep probabilistic dependencies in a stream by using a variable-order Markov model. By being able to look deeper into the past, we achieve higher accuracy scores compared to other state-of-the-art solutions for CEF, as shown in our extensive empirical analysis.
- Our framework can perform various types of forecasting and thus subsumes previous methods that restrict themselves to one type of forecasting. It can perform both simple event forecasting (i.e., predicting what the next input event might be) and Complex Event forecasting (events defined through a pattern). As we explain later, moving from simple event to Complex Event forecasting is not trivial. Using simple event forecasting to project in the future the most probable sequence of input events and then attempt to detect Complex Events on this future sequence yields sub-optimal results. A system that can perform simple event forecasting cannot thus be assumed to perform CEF as well.
- We also discuss the issue of how the forecasts of a CEF system may be evaluated with respect to their quality. Previous methods have used metrics borrowed from time-series forecasting (e.g., the root mean square error) or typical machine learning tasks (e.g., precision). We propose a more comprehensive set of metrics that takes into account the idiosyncrasies of CEF. Besides accuracy itself, the usefulness of forecasts is also judged by their "earliness". We discuss how the notion of earliness may be quantified.

We now present the general approach of CER/CEF systems, along with an example that we will use throughout the rest of the chapter to make our presentation more accessible.

The input to a CER system 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. Typically, CEs must be detected with very low latency, which, in certain cases, may even be in the order of a few milliseconds [56, 75, 93].

Table 6.1: Example event stream from the maritime domain.

Navigational status	fishing	fishing	fishing	under way	under way	under way	...
vessel id	78986	78986	78986	78986	78986	78986	...
speed	2	1	3	22	19	27	...
timestamp	1	2	3	4	5	6	...

■ **Example 6.1** As an example, consider the scenario of a system receiving an input stream consisting of events emitted from vessels sailing at sea. These events may contain information regarding the status of a vessel, e.g., its location, speed and heading. This is indeed a real-world scenario and the emitted messages are called AIS (Automatic Identification System) messages. Besides information about a vessel's kinematic behavior, each such message may contain additional information about the vessel's status (e.g., whether it is fishing), along with a timestamp and a unique vessel identifier. Table 6.1 shows a possible stream of AIS messages, including *speed* and *timestamp* information.

A maritime expert may be interested to detect several activity patterns for the monitored vessels, such as sudden changes in the kinematic behavior of a vessel (e.g., sudden accelerations), sailing in protected (e.g., NATURA) areas, etc. The typical workflow consists of the analyst first writing these patterns in some (usually) declarative language, which are then used by a computational model applied on the stream of SDEs to detect CEs. ■

The rest of the chapter is structured as follows. In Section 6.1 we discuss the formalism of symbolic automata and how it can be adapted to perform recognition on real-time event streams. Section 6.2 shows how we can create a probabilistic model for a symbolic automaton by using prediction suffix trees, while Section 6.3 presents a detailed complexity analysis. The chapter assumes a basic familiarity with automata theory, logic and Markov chains. In Table 6.1 we have gathered the notation that we use throughout the chapter, along with a brief description of every symbol.

6.1 Complex event recognition with symbolic automata

Our approach for CEF is based on a specific formal framework for CER, which we are presenting here. There are various surveys of CER methods, presenting various CER systems and languages [11, 42, 64]. Despite this fact though, there is still no consensus about which operators must be supported by a CER language and what their semantics should be. In this chapter, we follow [64] and [67], which have established some core operators that are most often used. In a spirit similar to [67], we use automata as our computational model and define a CER language whose expressions can readily be converted to automata. Instead of choosing one of the automaton models already proposed in the CER literature, we employ symbolic regular expressions and automata [43, 44, 134]. The rationale behind our choice is that, contrary to other automata-based CER models, symbolic regular expressions and automata have nice closure properties and clear (both declarative and operational), compositional semantics (see [67] for a similar line of work, based on symbolic transducers). In previous automata-based CER systems, it is unclear which operators may be used and if they can be arbitrarily combined (see [64, 67] for a discussion of this issue). On the contrary, the use of symbolic automata allows us to construct any pattern that one may desire through an arbitrary use of the provided operators. One can then check whether a stream satisfies some pattern either declaratively (by making use of the definition for symbolic expressions, presented below) or operationally (by using a symbolic automaton). This even allows us to write unit tests (as we have indeed done) ensuring that the semantics of symbolic regular expressions do indeed coincide with those of symbolic automata, something not possible with other frameworks. In previous methods, there is also a lack of understanding with respect to the properties of the employed computational models, e.g., whether the proposed automata are determinizable, an important feature for our work. Symbolic automata, on the other hand, have nice closure properties and are well-studied. Notice that this would also be an important feature for possible optimizations based on pattern re-writing, since such re-writing would require us to have a mechanism determining whether two expressions are equivalent. Our framework provides such a mechanism.

6.1.1 Symbolic expressions and automata

The main idea behind symbolic automata is that each transition, instead of being labeled with a symbol from an alphabet, is equipped with a unary formula from an effective

Table 6.2: Notation used throughout the chapter.

Symbol	Meaning
	Boolean algebra
$\mathscr{A}$	effective Boolean algebra
$\mathscr{D}$	domain elements of a Boolean algebra
Ψ	predicates of a Boolean algebra
$\bot, \top$	FALSE and TRUE predicates of a Boolean algebra
$\wedge$, $\vee$, $\neg$	logical conjunction, disjunction, negation
$\mathscr{L}$ ($\mathscr{L} \subseteq \mathscr{D}^*$)	a language over $\mathscr{D}$
	Symbolic expressions and automata
ϵ	the “empty” symbol
$R_1 + R_2$, $R_1 \cdot R_2$, R^*	regular disjunction / concatenation / iteration
M	automaton
Q, q^s, Q^f	automaton states / start state / final states
Δ, δ	automaton transition function / transition
$\mathscr{L}(R)$, $\mathscr{L}(M)$	language of expression R / automaton M
	Streaming expressions and automata
$t_i \in \mathscr{D}$	tuple / simple event
$S = t_1, t_2, \cdots$, $S_{i..j} = t_i, \cdots, t_j$	stream / stream “slice” from index i to j
$c = [i,q]$	automaton configuration (i current position, q current state)
$[i,q] \overset{\delta}{\rightarrow} [i',q']$	configuration succession
$\rho = [1,q_1] \overset{\delta_1}{\rightarrow} \cdots \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}]$	run of automaton M over stream $S_{1..k}$
$N = Minterms(Predicates(M))$	minterms of automaton M
	Variable-order Markov models
Σ, σ, s	alphabet of classical automaton / symbol / string
$suffix(s)$	the longest suffix of s different from s
$\hat{P}$	predictor
$l(\hat{P}, S_{1..k})$	average log-loss of $\hat{P}$ over $S_{1..k}$
T	prediction suffix tree
γ_s	next symbol probability function for a node of T
θ_1, θ_2	thresholds for tree learning
α	approximation parameter for tree learning
n	maximum number of states for learned suffix automaton
m	order of suffix tree / automaton / Markov model
τ	transition function of suffix automaton
γ, Γ	next symbol probability function of suffix automaton / embedding
π	initial probability distribution over start states of automaton / embedding

Boolean algebra [44]. A symbolic automaton can then read strings of elements and, upon reading an element while in a given state, can apply the predicates of this state's outgoing transitions to that element. The transitions whose predicates evaluate to TRUE are said to be "enabled" and the automaton moves to their target states.

The formal definition of an effective Boolean algebra is the following:

Definition 6.1.1 — Effective Boolean algebra (44). An effective Boolean algebra is a tuple $(\mathscr{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where

- $\mathscr{D}$ is a set of domain elements;
- Ψ is a set of predicates closed under the Boolean connectives;
- $\bot, \top \in \Psi$;
- and the component $[\![\_]\!] : \Psi \rightarrow 2^{\mathscr{D}}$ is a function such that
  - $[\![\bot]\!] = \emptyset$
  - $[\![\top]\!] = \mathscr{D}$
  - and $\forall \phi, \psi \in \Psi$:
    * $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$
    * $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$
    * $[\![\neg\phi]\!] = \mathscr{D} \setminus [\![\phi]\!]$

It is also required that checking satisfiability of ϕ, i.e., whether $[\![\phi]\!] \neq \emptyset$, is decidable and that the operations of $\vee$, $\wedge$ and $\neg$ are computable.

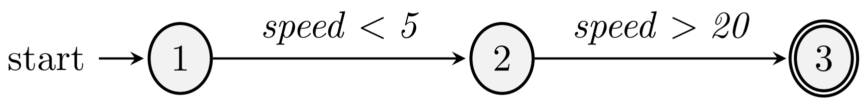
■ **Example 6.2** Using our running example, such an algebra could be one consisting of AIS messages, corresponding to $\mathscr{D}$, along with two predicates about the speed of a vessel, e.g., $speed < 5$ and $speed > 20$. These predicates would correspond to Ψ. The predicate $speed < 5$ would be mapped, via $[\![\_]\!]$, to the set of all AIS messages whose speed level is below 5 knots. According to the definition above, $\bot$ and $\top$ should also belong to Ψ, along with all the combinations of the original two predicates constructed from the Boolean connectives, e.g., $\neg(speed < 5) \wedge \neg(speed > 20)$. ■

Elements of $\mathscr{D}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathscr{L}$ constructed from elements of $\mathscr{D}$ ($\mathscr{L} \subseteq \mathscr{D}^*$, where * denotes Kleene-star) is called a language over $\mathscr{D}$.

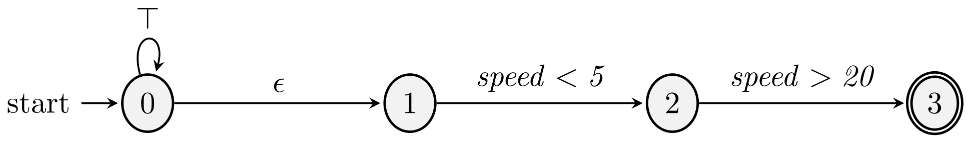
As with classical regular expressions [76], we can use symbolic regular expressions to represent a class of languages over $\mathscr{D}$.

Definition 6.1.2 — Symbolic regular expression. A symbolic regular expression (*SRE*) over an effective Boolean algebra $(\mathscr{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ is recursively defined as follows:

- The constants ε and $\emptyset$ are symbolic regular expressions with $\mathscr{L}(\varepsilon) = \{\varepsilon\}$ and $\mathscr{L}(\emptyset) = \{\emptyset\}$;
- If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathscr{L}(\psi) = [\![\psi]\!]$, i.e., the language of ψ is the subset of $\mathscr{D}$ for which ψ evaluates to TRUE;
- Disjunction / Union If R_1 and R_2 are symbolic regular expressions, then $R := R_1 + R_2$ is also a symbolic regular expression, with $\mathscr{L}(R) = \mathscr{L}(R_1) \cup \mathscr{L}(R_2)$;
- Concatenation / Sequence If R_1 and R_2 are symbolic regular expressions, then $R := R_1 \cdot R_2$ is also a symbolic regular expression, with $\mathscr{L}(R) = \mathscr{L}(R_1) \cdot \mathscr{L}(R_2)$, where $\cdot$ denotes concatenation. $\mathscr{L}(R)$ is then the set of all strings constructed from concatenating each element of $\mathscr{L}(R_1)$ with each element of $\mathscr{L}(R_2)$;
- Iteration / Kleene-star If R is a symbolic regular expression, then $R' := R^*$ is a



(a) *SFA* for the *SRE* $R := (speed < 5) \cdot (speed > 20)$.



(b) Streaming *SFA* for $R := (speed < 5) \cdot (speed > 20)$. $\top$ is a special predicate that always evaluates to TRUE. $\top$ transitions are thus triggered for every event. ε transitions triggered even in the absence of an event.

Figure 6.1: Examples of a symbolic automaton and streaming symbolic automaton. States with double circles are final.

symbolic regular expression, with $\mathscr{L}(R^*) = (\mathscr{L}(R))^*$, where $\mathscr{L}^* = \bigcup_{i \geq 0} \mathscr{L}^i$ and $\mathscr{L}^i$ is the concatenation of $\mathscr{L}$ with itself i times.

■ **Example 6.3** As an example, if we want to detect instances of a vessel accelerating suddenly, we could write the expression $R := (speed < 5) \cdot (speed > 20)$. The third and fourth events of the stream of Table 6.1 would then belong to the language of R. ■

Given a Boolean algebra, we can also define symbolic automata. The definition of a symbolic automaton is the following:

Definition 6.1.3 — Symbolic finite automaton [44]. A symbolic finite automaton (*SFA*) is a tuple $M = (\mathscr{A}, Q, q^s, Q^f, \Delta)$, where
- $\mathscr{A}$ is an effective Boolean algebra;
- Q is a finite set of states;
- $q^s \in Q$ is the initial state;
- $Q^f \subseteq Q$ is the set of final states;
- $\Delta \subseteq Q \times \Psi_{\mathscr{A}} \times Q$ is a finite set of transitions.

A string $w = a_1 a_2 \cdots a_k$ is accepted by a *SFA* M iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q^s$ and $q_k \in Q^f$. We refer to the set of strings accepted by M as the language of M, denoted by $\mathscr{L}(M)$ [44]. Figure 6.1a shows a *SFA* that can detect the expression of sudden acceleration for our running example.

As with classical regular expressions and automata, we can prove that every symbolic regular expression can be translated to an equivalent (i.e., with the same language) symbolic automaton.

Proposition 6.1.1 For every symbolic regular expression R there exists a symbolic finite automaton M such that $\mathscr{L}(R) = \mathscr{L}(M)$.

Proof. The proof is essentially the same as that for classical expressions and automata [76]. It is a constructive proof starting from the base case of an expression that is a single predicate (instead of a symbol, as in classical expressions) and then proceeds in a manner identical to that of the classical case. For the sake of completeness, the full proof is provided in the Appendix, Section A.1. ■

6.1.2 Streaming expressions and automata

Our discussion thus far has focused on how *SRE* and *SFA* can be applied to bounded strings that are known in their totality before recognition. We feed a string to a *SFA* and we expect an answer about whether the whole string belongs to the automaton's language or not. However, in CER and CEF we need to handle continuously updated streams of events and detect instances of *SRE* 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.

In order to accommodate this scenario, slight modifications are required so that *SRE* and *SFA* may work in a streaming setting. First, we need to make sure that the automaton can start its recognition after every new element. If we have a classical regular expression R, we can achieve this by applying on the stream the expression $\Sigma^* \cdot R$, where Σ is the automaton's (classical) alphabet. For example, if we apply $R := \{a, b, c\}^* \cdot (a \cdot b)$ on the stream a, b, c, a, b, c, the corresponding automaton would indeed reach its final state after reading the second and the fifth symbols. In our case, events come in the form of tuples with both numerical and categorical values. Using database systems terminology we can speak of tuples from relations of a database schema [67]. These tuples constitute the set of domain elements $\mathscr{D}$. A stream S then has the form of an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a tuple ($t_i \in \mathscr{D}$). Our goal is to report the indices i at which a CE 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 *SRE* R is detected at k iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in \mathscr{L}(R)$. In order to detect CEs of a *SRE* R on a stream, we use a streaming version of *SRE* and *SFA*.

Definition 6.1.4 — Streaming SRE and SFA. If R is a *SRE*, then $R_s = \top^* \cdot R$ is called the streaming *SRE* (*sSRE*) corresponding to R. A *SFA* M_{R_s} constructed from R_s is called a streaming *SFA* (*sSFA*) corresponding to R.

Using R_s we can detect CEs of R while reading a stream S, since a stream segment $S_{m..k}$ belongs to the language of R iff the prefix $S_{1..k}$ belongs to the language of R_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 6.1.2 If $S = t_1, t_2, \cdots$ is a stream of domain elements from an effective Boolean algebra $\mathscr{A} = (\mathscr{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, where $t_i \in \mathscr{D}$, and R is a symbolic regular expression over the same algebra, then, for every $S_{m..k}$, $S_{m..k} \in \mathscr{L}(R)$ iff $S_{1..k} \in \mathscr{L}(R_s)$ (and $S_{1..k} \in \mathscr{L}(M_{R_s})$).

Proof. The proof is provided in the Appendix, Section A.2. ■

■ **Example 6.4** As an example, if $R := (speed < 5) \cdot (speed > 20)$ is the pattern for sudden acceleration, then its *sSRE* would be $R_s := \top^* \cdot (speed < 5) \cdot (speed > 20)$. After reading the fourth event of the stream of Table 6.1, $S_{1..4}$ would belong to the language of $\mathscr{L}(R_s)$ and $S_{3..4}$ to the language of $\mathscr{L}(R)$. Figure 6.1b shows an example *sSFA*. ■

Note that *sSRE* and *sSFA* are just special cases of *SRE* and *SFA* respectively. Therefore, every result that holds for *SRE* and *SFA* also holds for *sSRE* and *sSFA* as well.

The streaming behavior of a *sSFA* as it consumes a stream S can be formally defined using the notion of configuration:

Definition 6.1.5 — Configuration of sSFA. Assume $S = t_1, t_2, \cdots$ is a stream of domain elements from an effective Boolean algebra, R a symbolic regular expression over the same algebra and M_{R_s} a *sSFA* corresponding to R. A configuration c of M_{R_s} is a tuple $[i, q]$, where i is the current position of the stream, i.e., the index of the next event to be consumed, and q the current state of M_{R_s}. We say that $c' = [i', q']$ is a successor of c iff:

- $\exists \delta \in M_{R_s}.\Delta : \delta = (q, \psi, q') \wedge (t_i \in [\![\psi]\!] \vee \psi = \varepsilon)$;
- $i = i'$ if $\delta = \varepsilon$. Otherwise, $i' = i + 1$.

We denote a succession by $[i, q] \overset{\delta}{\rightarrow} [i', q']$.

For the initial configuration c^s, before consuming any events, we have that $i = 1$ and $c^s.q = M_{R_s}.q^s$, i.e. the state of the first configuration is the initial state of M_{R_s}. In other words, for every index i, we move from our current state q to another state q' if there is an outgoing transition from q to q' and the predicate on this transition evaluates to TRUE for t_i. We then increase the reading position by 1. Alternatively, if the transition is an ε-transition, we move to q' without increasing the reading position.

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

Definition 6.1.6 — Run of sSFA over stream. A run ρ of a *sSFA* M over a stream $S_{1..k}$ is a sequence of successor configurations $[1, q_1 = M.q^s] \overset{\delta_1}{\rightarrow} [2, q_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}]$. A run is called accepting iff $q_{k+1} \in M.Q^f$.

A run ρ of a *sSFA* M_{R_s} over a stream $S_{1..k}$ is accepting iff $S_{1..k} \in \mathscr{L}(R_s)$, since M_{R_s}, after reading $S_{1..k}$, must have reached a final state. Therefore, for a *sSFA* that consumes a stream, the existence of an accepting run with configuration index $k+1$ implies that a CE for the *SRE* R has been detected at the stream index k.

As far as the temporal model is concerned, we assume that all SDEs are instantaneous. They all carry a *timestamp* attribute which is single, unique numerical value. We also assume that the stream of SDEs is temporally sorted. A sequence/concatenation operator is thus satisfied if the event of its first operand precedes in time the event of its second operand. The exception to the above assumptions is when the stream is composed of multiple partitions and the defined pattern is applied on a per-partition basis. For example, in the maritime domain a stream may be composed of the sub-streams generated by all vessels and we may want to detect the same pattern for each individual vessel. In such cases, the above assumptions must hold for each separate partition but not necessarily across all partitions. Another general assumption is that there is no imposed limit on the time elapsed between consecutive events in a sequence operation.

6.1.3 Expressive power of symbolic regular expressions

We conclude this section with some remarks about the expressive power of *SRE* and *SFA* and how it meets the requirements of a CER system. As discussed in [64, 67], besides the three operators of regular expressions that we have presented and implemented (disjunction, sequence, iteration), there exist some extra operators which should be supported by a CER system. *Negation* is one them. If we use ! to denote the negation operator, then $R' := !R$ defines a language which is the complement of the language of R. Since *SFA* are closed under complement [44], negation is an operator that can be supported by our framework and has also been implemented (but not discussed further).

The same is true for the operator of *conjunction*. If we use $\wedge$ to denote conjunction, then $R := R_1 \wedge R_2$ is an expression whose language consists of concatenated elements of $\mathscr{L}(R_1)$

and $\mathscr{L}(R_2)$, regardless of their order, i.e., $\mathscr{L}(R) = \mathscr{L}(R_1) \cdot \mathscr{L}(R_2) \cup \mathscr{L}(R_2) \cdot \mathscr{L}(R_1)$. This operator can thus be equivalently expressed using the already available operators of concatenation $(\cdot)$ and disjunction $(+)$.

Another important notion in CER is that of *selection policies*. An expression like $R := R_1 \cdot R_2$ typically implies that an instance of R_2 must immediately follow an instance of R_1. As a result, for the stream of Table 6.1 and $R := (speed < 5) \cdot (speed > 20)$, only one match will be detected at $timestamp = 4$. With selection policies, we can relax the requirement for contiguous instances.

■ **Example 6.5** For example, with the so-called skip-till-any-match policy, any number of events are allowed to occur between R_1 and R_2. If we apply this policy on $R := (speed < 5) \cdot (speed > 20)$, we would detect six CEs, since the first three events of Table 6.1 can be matched with the two events at $timestamp = 4$ and at $timestamp = 6$, if we ignore all intermediate events. ■

Selection policies can also be accommodated by our framework and have been implemented. For a proof, using symbolic transducers, see [67]. Notice, for example, that an expression $R := R_1 \cdot R_2$ can be evaluated with skip-till-any-match by being rewritten as $R' := R_1 \cdot \top^* \cdot R_2$, so that any number of events may occur between R_1 and R_2.

Support for hierarchical definition of patterns, i.e., the ability to define patterns in terms of other patterns, is yet another important feature in many CER systems. Since *SRE* and *SFA* are compositional by nature, hierarchies are supported by default in our framework. Although we do not treat these operators and functionalities explicitly here, their incorporation is possible within the expressive limits of *SRE* and *SFA* and the results that we present in the next sections would still hold.

6.2 Building a probabilistic model

The main idea behind our forecasting method is the following: Given a pattern R in the form of a *SRE*, we first construct a *sSFA* as described in the previous section. For event recognition, this would already be enough, but in order to perform event forecasting, we translate the *sSFA* to an equivalent deterministic *SFA* (*DSFA*). This *DSFA* can then be used to learn a probabilistic model, typically a Markov chain, that encodes dependencies among the events in an input stream. Note that a non-deterministic automaton cannot be directly converted to a Markov chain, since from each state we might be able to move to multiple other target states with a given event. Therefore, we first determinize the automaton.

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 CE encoded by the automaton. The issue that we address in this chapter is how to build a model which retains long-term dependencies that are useful for forecasting.

Figure 6.2 depicts all the required steps in order to produce forecasts for a given pattern. We have already described steps 1 and 2. In Section 6.2.1 we describe step 3. In Sections 6.2.2 - 6.2.4 we present step 4, our proposed method for constructing a probabilistic model for a pattern, based on prediction suffix trees. Steps 5 and 6 are described in Section 6.2.4. After learning a model, we first need to estimate the so-called *waiting-time distributions* for each state of our automaton. Roughly speaking, 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 CE has a high probability of occurring. Finally, Section 14

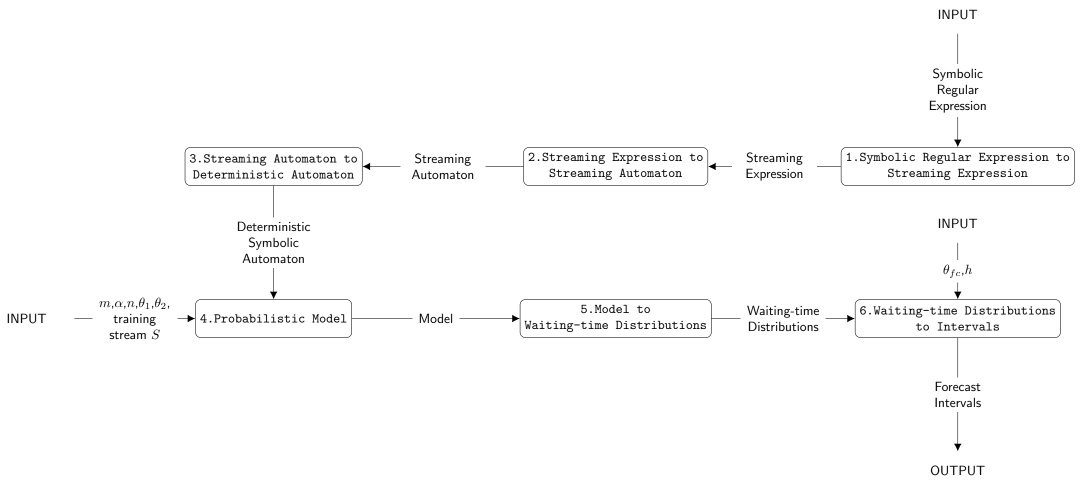

INPUT
Symbolic Regular Expression
1.Symbolic Regular Expression to Streaming Expression
Streaming Expression
2.Streaming Expression to Streaming Automaton
Streaming Automaton
3.Streaming Automaton to Deterministic Automaton
Deterministic Symbolic Automaton
INPUT
m,α,n,θ_1,θ_2, training stream S
4.Probabilistic Model
Model
5.Model to Waiting-time Distributions
Waiting-time Distributions
INPUT
θ_{fc},h
6.Waiting-time Distributions to Intervals
Forecast Intervals
OUTPUT

Figure 6.2: Steps for calculating the forecast intervals of a given pattern.

discusses an optimization that allows us to bypass the explicit construction of the Markov chain and Section 6.3 presents a full complexity analysis.

6.2.1 Deterministic symbolic automata

The definition of *DSFA* is similar to that of classical deterministic automata. Intuitively, we require that, for every state and every tuple/character, the *SFA* can move to at most one next state upon reading that tuple/character. We note though that it is not enough to require that all outgoing transitions from a state have different predicates as guards. Symbolic automata differ from classical in one important aspect. For the latter, if we start from a given state and we have two outgoing transitions with different labels, then it is not possible for both of these transition to be triggered simultaneously (i.e., with the same character). For symbolic automata, on the other hand, two predicates may be different but still both evaluate to TRUE for the same tuple and thus two transitions with different predicates may both be triggered with the same tuple. Therefore, the formal definition for *DSFA* must take this into account:

Definition 6.2.1 — Deterministic SFA (44). A *SFA* M is deterministic if, for all transitions $(q,\psi_1,q_1),(q,\psi_2,q_2) \in M.\Delta$, if $q_1 \neq q_2$ then $[\![\psi_1 \wedge \psi_2]\!] = \emptyset$.

Using this definition for *DSFA* it can be proven that *SFA* are indeed closed under determinization [44]. The determinization process first needs to create the *minterms* of the predicates of a *SFA* M, i.e., the set of maximal satisfiable Boolean combinations of such predicates, denoted by $Minterms(Predicates(M))$, and then use these minterms as guards for the *DSFA* [44].

There are two factors that can lead to a combinatorial explosion of the number of states of the resulting *DSFA*: first, the fact that the powerset of the states of the original *SFA* must be constructed (similarly to classical automata); second, the fact that the number of minterms (and, thus, outgoing transitions from each *DSFA* state) is an exponential function of the number of the original *SFA* predicates. In order to mitigate this doubly exponential cost, we follow two simple optimization techniques. As is typically done with classical automata as well, instead of constructing the powerset of states of the *SFA* and then adding transitions, we construct the states of the *DSFA* incrementally, starting from its initial state,

Table 6.3: The set of simplified minterms for the predicates $\psi_A := speed < 5$ and $\psi_B := speed > 20$.

Original	Simplified	Reason
$\psi_A \wedge \psi_B$	discard	unsatisfiable
$\psi_A \wedge \neg\psi_B$	ψ_A	$\psi_A \vDash \neg\psi_B$
$\neg\psi_A \wedge \psi_B$	ψ_B	$\psi_B \vDash \neg\psi_B$
$\neg\psi_A \wedge \neg\psi_B$	$\neg\psi_A \wedge \neg\psi_B$	for events whose speed is between 5 and 20

without adding states that will be inaccessible in the final *DSFA*. We can also reduce the number of minterms by taking advantage of some previous knowledge about some of the predicates that we might have. In cases where we know that some of the predicates are mutually exclusive, i.e., at most one of them can evaluate to TRUE, then we can both discard some minterms and simplify some others.

■ **Example 6.6** For example, if we have two predicates, $\psi_A := speed < 5$ and $\psi_B := speed > 20$, then we also know that ψ_A and ψ_B are mutually exclusive. As a result, we can simplify the minterms, as shown in Table 6.3. ■

Before moving to the discussion about how a *DSFA* can be converted to a Markov chain, we present a useful lemma. We will show that a *DSFA* always has an equivalent (through an isomorphism) deterministic classical automaton. 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.

First note that $Minterms(Predicates(M))$ induces a finite set of equivalence classes on the (possibly infinite) set of domain elements of M [44].

■ **Example 6.7** For example, if $Predicates(M) = \{\psi_1, \psi_2\}$, then $Minterms(Predicates(M)) = \{\psi_1 \wedge \psi_2, \psi_1 \wedge \neg\psi_2, \neg\psi_1 \wedge \psi_2, \neg\psi_1 \wedge \neg\psi_2\}$, and we can map each domain element, which, in our case, is a tuple, to exactly one of these 4 minterms: the one that evaluates to TRUE when applied to the element. ■

Similarly, the set of minterms induces a set of equivalence classes on the set of strings (event streams in our case).

■ **Example 6.8** For example, if $S{=}t_1, \cdots, t_k$ is an event stream, then it could be mapped to $S'{=}a, \cdots, b$, with a corresponding to $\psi_1 \wedge \neg\psi_2$ if $\psi_1(t_1) \wedge \neg\psi_2(t_1) = \text{TRUE}$, b to $\psi_1 \wedge \psi_2$, etc. ■

Definition 6.2.2 — Stream induced by the minterms of a *DSFA*. If S is a stream from the domain elements of the algebra of a *DSFA* M and $N = Minterms(Predicates(M))$, then the stream S' induced by applying N on S is the equivalence class of S induced by N.

We can now prove the lemma, which states that for every *DSFA* there exists an equivalent classical deterministic automaton.

Lemma 6.2.1 For every deterministic symbolic finite automaton (*DSFA*) M_s there exists a deterministic classical finite automaton (*DFA*) M_c such that $\mathscr{L}(M_c)$ is the set of strings induced by applying $N = Minterms(Predicates(M_s))$ to $\mathscr{L}(M_s)$.

Proof. From an algebraic point of view, the set $N = Minterms(Predicates(M))$ may be treated as a generator of the monoid N^*, with concatenation as the operation. If the cardinality of N is k, then we can always find a set $\Sigma = \{a_1, \cdots, a_k\}$ of k distinct symbols and then a morphism (in fact, an isomorphism) $\phi : N^* \rightarrow \Sigma^*$ that maps each minterm to exactly one, unique a_i. A classical *DFA* M_c can then be constructed by relabeling the *DSFA* M_s under ϕ, i.e., by copying/renaming the states and transitions of the original *DSFA* M_s and by replacing the label of each transition of M_s by the image of this label under ϕ. Then, the behavior of M_c (the language it accepts) is the image under ϕ of the behavior of M_s [122]. Or, equivalently, the language of M_c is the set of strings induced by applying $N = Minterms(Predicates(M_s))$ to $\mathscr{L}(M_s)$. ■

A direct consequence drawn from the proof of the above lemma is that, for every run $\rho = [1, q_1] \overset{\delta_1}{\rightarrow} [2, q_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}]$ followed by a *DSFA* M_s by consuming a symbolic string (stream of tuples) S, the run that the equivalent *DFA* M_c follows by consuming the induced string S' is also $\rho' = [1, q_1] \overset{\delta_1}{\rightarrow} [2, q_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_k}{\rightarrow} [k+1, q_{k+1}]$, i.e., M_c follows the same copied/renamed states and the same copied/relabeled transitions.

This direct relationship between *DSFA* and classical *DFA* allows us to transfer techniques developed for classical *DFA* to the study of *DSFA*. Moreover, we can simplify our notation by employing the terminology of symbols/characters and strings/words that is typical for classical automata. Henceforth, we will be using symbols and strings as in classical theories of automata and strings (simple lowercase letters to denote symbols), but the reader should bear in mind that, in our case, each symbol always corresponds to a predicate and, more precisely, to a minterm of a *DSFA*.

■ **Example 6.9** For example, the symbol a may actually refer to the minterm $(speed < 5) \wedge (speed > 20)$, the symbol b to $(speed < 5) \wedge \neg(speed > 20)$, etc. ■

6.2.2 Variable-order Markov models

Assuming that we have a deterministic automaton, the next question is how we can build a probabilistic model that captures the statistical properties of the streams to be processed by this automaton. With such a model, we could then make inferences about the automaton's expected behavior as it reads event streams. One approach would be to map each state of the automaton to a state of a Markov chain, then apply the automaton on a training stream of symbols, count the number of transitions from each state to every other target state and use these counts to calculate the transition probabilities. This is the approach followed in [101].

However, there is an important issue with the way in which this approach models transition probabilities. Namely, a probability is attached to the transition between two states, say state 1 and state 2, ignoring the way in which state 1 has been reached, i.e., failing to capture the sequence of symbols.

■ **Example 6.10** For example, in Figure 6.3, state 0 can be reached after observing symbol b or symbol c. The outgoing transition probabilities do not distinguish between the two cases. Instead, they just capture the probability of a given that the previous symbol was b or c. This introduces ambiguity and if there are many such states in the automaton, we may end up with a Markov chain that is first-order (with respect to its states), but nevertheless provides no memory of the stream itself. It may be unable to capture first-order (or higher order) dependencies in the stream of events. In the worst case (if every state can be reached

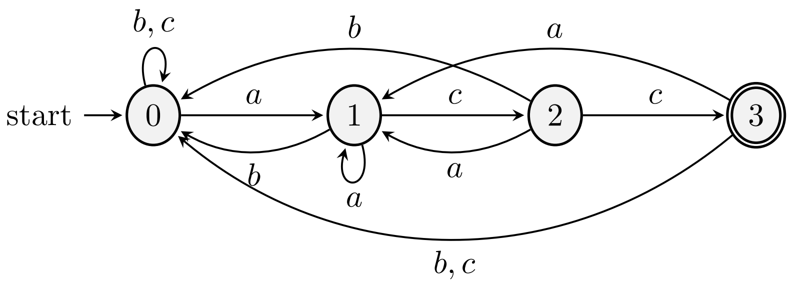


Figure 6.3: A classical automaton for the expression $R := a \cdot c \cdot c$ with alphabet $\Sigma = \{a, b, c\}$. State 1 can always remember the last symbol seen, since it can be reached only with a. State 0 can be reached with b or c.

with any symbol), such a Markov chain may essentially assume that the stream is composed of i.i.d. events. ∎

An alternative approach, followed in [6, 8], is to first set a maximum order m that we need to capture and then iteratively split each state of the original automaton into as many states as required so that each new state can remember the past m symbols that have led to it. The new automaton that results from this splitting process is equivalent to the original, in the sense that they recognize the same language, but can always remember the last m symbols of the stream. With this approach, it is indeed possible to guarantee that m-order dependencies can be captured. As expected though, higher values of m can quickly lead to an exponential growth of the number of states and the approach may be practical only for low values of m.

We propose the use of a variable-order Markov model (VMM) to mitigate the high cost of increasing the order m [24, 28, 37, 120, 121, 140]. This allows us to increase m to values not possible with the previous approaches and thus capture longer-term dependencies, which can lead to a better accuracy. An alternative would be to use hidden Markov models (HMMs) [118], which are generally more expressive than bounded-order (either full or variable) Markov models. However, HMMs often require large training datasets [2, 24]. Another problem is that it is not always obvious how a domain can be modeled through HMMs and a deep understanding of the domain may be required [24]. The relation between an automaton and the observed state of a HMM is not straightforward and it is not evident how a HMM would capture an automaton's behavior.

Different Markov models of variable order have been proposed in the literature (see [24] for a nice comparative study). The general approach of such models is as follows: let Σ 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 $l(\hat{P}, S_{1..k}) = -\frac{1}{T}\sum_{i=1}^{k} log\hat{P}(t_i \mid t_1 \cdots t_{i-1})$. Minimizing the log-loss is equivalent to maximizing the likelihood $\hat{P}(S_{1..k}) = \prod_{i=1}^{k} \hat{P}(t_i \mid t_1 \ldots t_{i-1})$. The average log-loss may also be viewed as a measure of the average compression rate achieved on the test sequence [24]. 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 full-order Markov models, the predictor $\hat{P}$ is derived through the estimation of conditional distributions $\hat{P}(\sigma \mid s)$, with m constant and equal to the assumed order of the Markov model. Variable-order Markov Models (VMMs), on the other hand, relax the assumption of m being fixed. The length of the "context" s (as is usually called) 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.

6.2.3 Prediction suffix trees

We use Prediction Suffix Trees (*PST*), as described in [120, 121], as our VMM of choice. The reason is that, once a *PST* has been learned, it can be readily converted to a probabilistic automaton. More precisely, we learn a probabilistic suffix automaton (*PSA*), whose states correspond to contexts of variable length. The outgoing transitions from each state of the *PSA* encode the conditional distribution of seeing a symbol given the context of that state. As we will show, this probabilistic automaton (or the tree itself) can then be combined with a symbolic automaton in a way that allows us to infer when a CE is expected to occur.

The formal definition of a PST is the following:

Definition 6.2.3 — Prediction Suffix Tree [121]. Let Σ be an alphabet. A PST T over Σ is a tree whose edges are labeled by symbols $\sigma \in \Sigma$ and each internal node has exactly one edge for every $\sigma \in \Sigma$ (hence, the degree is $\mid \Sigma \mid$). Each node is labeled by a pair (s, γ_s), where s is the string associated with the walk starting from that node and ending at the root, and $\gamma_s : \Sigma \rightarrow [0,1]$ is the next symbol probability function related with s. For every string s labeling a node, $\sum_{\sigma \in \Sigma} \gamma_s(\sigma) = 1$. The depth of the tree is its order m.

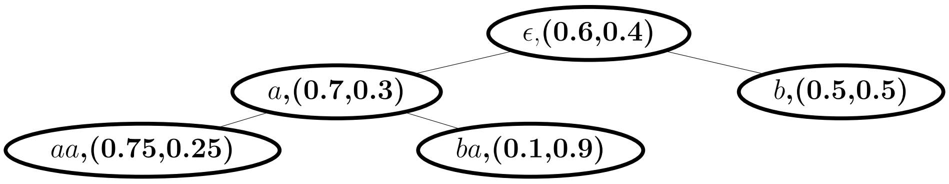
■ **Example 6.11** Figure 6.4a shows an example of a *PST* of order $m = 2$. According to this tree, if the last symbol that we have encountered in a stream is a and we ignore any other symbols that may have preceded it, then the probability of the next input symbol being again a is 0.7. However, we can obtain a better estimate of the next symbol probability by extending the context and looking one more symbol deeper into the past. Thus, if the last two symbols encountered are b, a, then the probability of seeing a again is very different (0.1). On the other hand, if the last symbol encountered is b, the next symbol probability distribution is $(0.5, 0.5)$ and, since the node $b, (0.5, 0.5)$ has not been expanded, this implies that its children would have the same distribution if they had been created. Therefore, the past does not affect the prediction and will not be used. ■

Note that a *PST* whose leaves are all of equal depth m corresponds to a full-order Markov model of order m, as its paths from the root to the leaves correspond to every possible context of length m.

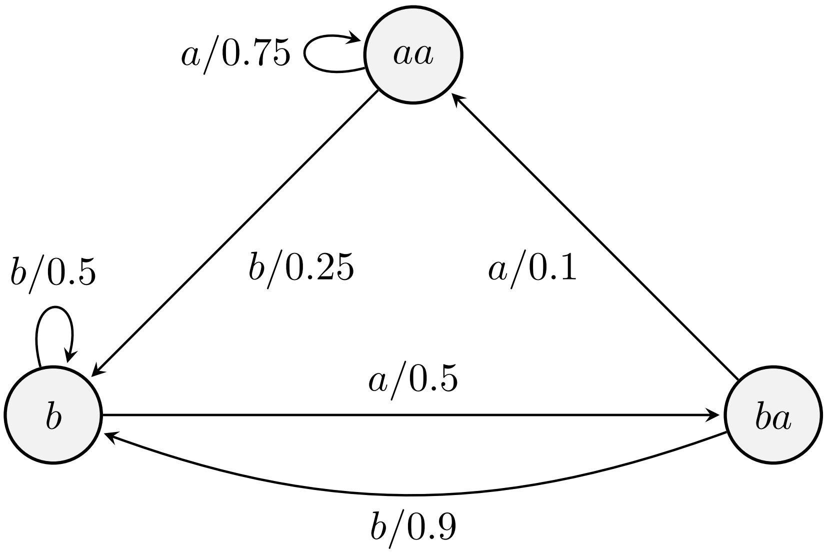
Our goal is to incrementally learn a *PST* $\hat{T}$ by adding new nodes only when it is necessary and then use $\hat{T}$ to construct a *PSA* $\hat{M}$ that will approximate the actual *PSA* M that has generated the training data. Assuming that we have derived an initial predictor $\hat{P}$ (as described in more detail in Section 6.2.5), the learning algorithm in [121] starts with a tree having only a single node, corresponding to the empty string ε. Then, it decides whether to add a new context/node s by checking two conditions:

- First, there must exist $\sigma \in \Sigma$ such that $\hat{P}(\sigma \mid s) > \theta_1$ must hold, i.e., σ must appear "often enough" after the suffix s;
- Second, $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} > \theta_2$ (or $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} < \frac{1}{\theta_2}$) must hold, i.e., it is "meaningful enough" to expand to s because there is a significant difference in the conditional probability of σ given s with respect to the same probability given the shorter context $suffix(s)$, where $suffix(s)$ is the longest suffix of s that is different from s.

The thresholds θ_1 and θ_2 depend, among others, on parameters α, n and m, α being



(a) Example *PST* T for $\Sigma = \{a, b\}$ and $m = 2$. Each node contains the label and the next symbol probability distribution for a and b.



(b) Example *PSA* M_S constructed from the tree T. Each state contains its label. Each transition is composed of the next symbol to be encountered along with that symbol's probability.

Figure 6.4: Example of a prediction suffix tree and its corresponding probabilistic suffix automaton.

an approximation parameter, measuring how close we want the estimated *PSA* $\hat{M}$ to be compared to the actual *PSA* M, n denoting the maximum number of states that we allow $\hat{M}$ to have and m denoting the maximum order/length of the dependencies we want to capture.

■ **Example 6.12** For example, consider node a in Figure 6.4a and assume that we are at a stage of the learning process where we have not yet added its children, aa and ba. We now want to check whether it is meaningful to add ba as a node. Assuming that the first condition is satisfied, we can then check the ratio $\frac{\hat{P}(\sigma|s)}{\hat{P}(\sigma|suffix(s))} = \frac{\hat{P}(a|ba)}{\hat{P}(a|a)} = \frac{0.1}{0.7} \approx 0.14$. If $\theta_2 = 1.05$, then $\frac{1}{\theta_2} \approx 0.95$ and the condition is satisfied, leading to the addition of node ba to the tree. ■

For more details, see [121].

Once a *PST* $\hat{T}$ has been learned, we can convert it to a *PSA* $\hat{M}$. The definition for *PSA* is the following:

Definition 6.2.4 — Probabilistic Suffix Automaton [121]. A Probabilistic Suffix Automaton M is a tuple $(Q, \Sigma, \tau, \gamma, \pi)$, where:

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\tau : Q \times \Sigma \rightarrow Q$ is the transition function;

- $\gamma : Q \times \Sigma \rightarrow [0,1]$ is the next symbol probability function;
- $\pi : Q \rightarrow [0,1]$ is the initial probability distribution over the starting states;

The following conditions must hold:

- For every $q \in Q$, it must hold that $\sum_{\sigma \in \Sigma} \gamma(q,\sigma) = 1$ and $\sum_{q \in Q} \pi(q) = 1$;
- Each $q \in Q$ is labeled by a string $s \in \Sigma^*$ and the set of labels is suffix free, i.e., no label s is a suffix of another label s';
- For every two states $q_1, q_2 \in Q$ and for every symbol $\sigma \in \Sigma$, if $\tau(q_1,\sigma) = q_2$ and q_1 is labeled by s_1, then q_2 is labeled by s_2, such that s_2 is a suffix of $s_1 \cdot \sigma$;
- For every s labeling some state q, and every symbol σ for which $\gamma(q,\sigma) > 0$, there exists a label which is a suffix of $s \cdot \sigma$;
- Finally, the graph of M is strongly connected.

Note that a *PSA* is a Markov chain. τ and γ can be combined into a single function, ignoring the symbols, and this function, together with the first condition of Definition 6.2.4, would define the transition matrix of a Markov chain. The last condition about M being strongly connected also ensures that the Markov chain is composed of a single recurrent class of states.

■ **Example 6.13** Figure 6.4b shows an example of a *PSA*, the one that we construct from the *PST* of Figure 6.4a, using the leaves of the tree as automaton states. A full-order *PSA* for $m = 2$ would require a total of 4 states, given that we have two symbols. If we use the *PST* of Figure 6.4a, we can construct the *PSA* of Figure 6.4b which has 3 states. State b does not need to be expanded to states bb and ab, since the tree tells us that such an expansion is not statistically meaningful. ■

Using a *PSA* we can process a stream of symbols and at every point be able to provide an estimate about the next symbols that will be encountered along with their probabilities. The state of the *PSA* at every moment corresponds to a suffix of the stream.

■ **Example 6.14** For example, according to the *PSA* of Figure 6.4b, if the last symbol consumed from the stream is b, then the *PSA* would be in state b and the probability of the next symbol being a would be 0.5. If the last symbol in the stream is a, we would need to expand this suffix to look at one more symbol in the past. If the last two symbols are aa, then the *PSA* would be in state aa and the probability of the next symbol being a again would be 0.75. ■

Note that a *PSA* does not act as an acceptor (there are no final states), but can act as a generator of strings. It can use π, its initial distribution on states, to select an initial state and generate its label as a first string and then continuously use γ to generate a symbol, move to a next state and repeat the same process. At every time, the label of its state is always a suffix of the string generated thus far. A *PSA* may also be used to read a string or stream of symbols. In this mode, the state of the *PSA* at every moment corresponds again to a suffix of the stream and the *PSA* can be used to calculate the probability of seeing any given string in the future, given the label of its current state. Our intention is to use this derived *PSA* to process streams of symbols, so that, while consuming a stream $S_{1..k}$, we can know what its meaningful suffix and use that suffix for any inferences.

R However, there is a subtle technical issue about the convertibility of a *PST* to a *PSA*. Not every *PST* can be converted to a *PSA* (but every *PST* can be converted to a larger class of so-call probabilistic automata). This is achievable under a certain condition. If this condition does not hold, then the *PST* can be converted to an automaton that is composed of a *PSA* as usual, with the addition of some extra states. These states, viewed as states of a Markov chain, are transient. This means that the automaton will move through these states for some transitions, but it will finally end into the states of the *PSA*, stay in that class and never return to any of the transient states. In fact, if the automaton starts in any of the transient states, then it will enter the single, recurrent class of the *PSA* in at most m transitions. Given the fact that in our work we deal with streams of infinite length, it is certain that, while reading a stream, the automaton will have entered the *PSA* after at most m symbols. Thus, instead of checking this condition, we prefer to simply construct only the *PSA* and wait (for at most m symbols) until the first $k \leq m$ symbols of a stream have been consumed and are equal to a label of the *PSA*. At this point, we set the current state of the *PSA* to the state with that label and start processing.

The above discussion seems to suggest that a *PSA* is constructed from the leaves of a *PST*. Thus, it should be expected that the number of states of a *PSA* should always be smaller than the total number of nodes of its *PST*. However, this is not true in the general case. In fact, in some cases the *PST* nodes might be significantly less than the *PSA* states. The reason is that a *PST*, as is produced by the learning algorithm described previously, might not be sufficient to construct a *PSA*. To remedy this situation, we need to expand the original *PST* $\hat{T}$ by adding more nodes in order to get a suitable *PST* $\hat{T}'$ and then construct the *PSA* from $\hat{T}'$. The leaves of $\hat{T}'$ (and thus the states of the *PSA*) could be significantly more than the leaves of $\hat{T}$. This issue is further discussed in Section 14.

6.2.4 Emitting forecasts

Our ultimate goal is to use the statistical properties of a stream, as encoded in a *PST* or a *PSA*, in order to infer when a Complex Event (CE) with a given Symbolic Regular Expression (*SRE*) R will be detected. Equivalently, we are interested in inferring when the *SFA* of R will reach one of its final states. To achieve this goal, we work as follows. We start with a *SRE* R and a training stream S. We first use R to construct an equivalent *sSFA* and then determinize this *sSFA* into a *DSFA* M_R. M_R can be used to perform recognition on any given stream, but cannot be used for probabilistic inference. Next, we use the minterms of M_R (acting as "symbols", see Lemma 6.2.1) and the training stream S to learn a *PST* T and (if required) a *PSA* M_S which encode the statistical properties of S. These probabilistic models do not yet have any knowledge of the structure of R (they only know its minterms), are not acceptors (the *PSA* does not have any final states) and cannot be used for recognition. We therefore need to combine the learned probabilistic model (T or M_S) with the automaton used for recognition (M_R).

At this point, there is a trade-off between memory and computation efficiency. If the online performance of our system is critical and we are not willing to make significant sacrifices in terms of computation efficiency, then we should combine the recognition automaton M_R with the *PSA* M_S. Using the *PSA* we can have a very efficient solution with minimal overhead on throughput. The downside of this approach is its memory footprint, which may limit the order of the model. Although we may increase the order beyond what is possible with full-order models, we may still not achieve the desired values, due to the significant memory requirements. Hence, if high accuracy and thus high order values are necessary, then we should combine the recognition automaton M_R directly with the *PST* T, bypassing the construction of the *PSA*. In practice prediction suffix trees often turn out to be more compact and memory efficient than probabilistic suffix automata, but trees need to be constantly traversed from root to leaves whereas an automaton simply needs to find the triggered transition and immediately jump to the next state. In the remainder of this Section, we present these two alternatives.

Using a Probabilistic Suffix Automaton (*PSA*)

We can combine a recognition automaton M_R and a *PSA* M_S into a single automaton M that has the power of both and can be used for recognizing and for forecasting occurrences of CEs of the expression R. We call M the *embedding* of M_S in M_R. The reason for merging the two automata is that we need to know at every point in time the state of M_R in order to estimate which future paths might actually lead to a final state (and thus a complex event). If only SDE forecasting was required, this merging would not be necessary. We could use M_R for recognition and then M_S for SDE forecasting. In our case, we need information about the structure of the pattern automaton and its current state to determine if and when it might reach a final state. The formal definition of an embedding is given below, where, in order to simplify notation, we use Lemma 6.2.1 and represent *DSFA* as classical deterministic automata.

Definition 6.2.5 — Embedding of a *PSA* in a *DSFA*. Let M_R be a *DSFA* (actually its mapping to a classical automaton) and M_S a *PSA* with the same alphabet. An embedding of M_S in M_R is a tuple $M = (Q, Q^s, Q^f, \Sigma, \Delta, \Gamma, \pi)$, where:

- Q is a finite set of states;
- $Q^s \subseteq Q$ is the set of initial states;
- $Q^f \subseteq Q$ is the set of final states;
- Σ is a finite alphabet;
- $\Delta : Q \times \Sigma \rightarrow Q$ is the transition function;
- $\Gamma : Q \times \Sigma \rightarrow [0,1]$ is the next symbol probability function;
- $\pi : Q \rightarrow [0,1]$ is the initial probability distribution.

The language $\mathscr{L}(M)$ of M is defined, as usual, as the set of strings that lead M to a final state. The following conditions must hold, in order for M to be an embedding of M_S in M_R:

- $\Sigma = M_R.\Sigma = M_S.\Sigma$;
- $\mathscr{L}(M) = \mathscr{L}(M_R)$;
- For every string/stream $S_{1..k}$, $P_M(S_{1..k}) = P_{M_S}(S_{1..k})$, where P_M denotes the probability of a string calculated by M (through Γ) and P_{M_S} the probability calculated by M_S (through γ).

The first condition ensures that all automata have the same alphabet. The second ensures that M is equivalent to M_R by having the same language. The third ensures that M is also equivalent to M_S, since both automata return the same probability for every string.

It can be shown that such an equivalent embedding can indeed be constructed for every *DSFA* and *PSA*.

Theorem 6.2.2 For every *DSFA* M_R and *PSA* M_S constructed using the minterms of M_R, there exists an embedding of M_S in M_R.

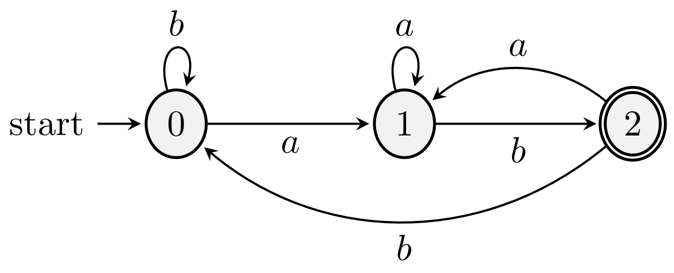
Proof. Construct an embedding in the following straightforward manner: First let its states be the Cartesian product $M_R.Q \times M_S.Q$, i.e., for every $q \in Q$, $q = (r,s)$ and $r \in M_R.Q$, $s \in M_S.Q$. Set the initial states of M as follows: for every $q = (r,s)$ such that $r = M_R.q^s$, set $q \in Q^s$. Similarly, for the final states, for every $q = (r,s)$ such that $r \in M_R.Q^f$, set $q \in Q^f$. Then let the transitions of M be defined as follows: A transition $\delta((r,s),\sigma) = (r^{'},s^{'})$ is added to M if there exists a transition $\delta_R(r,\sigma) = r^{'}$ in M_R and a transition $\tau(s,\sigma) = s^{'}$ in M_S. Let also Γ be defined as follows: $\Gamma((r,s),\sigma) = \gamma(s,\sigma)$. Finally, for the initial state distribution, we set:

$$\pi((r,s)) = \begin{cases} M_S.\pi(s) & \text{if } r = M_R.q^s \\ 0 & \text{otherwise} \end{cases}$$

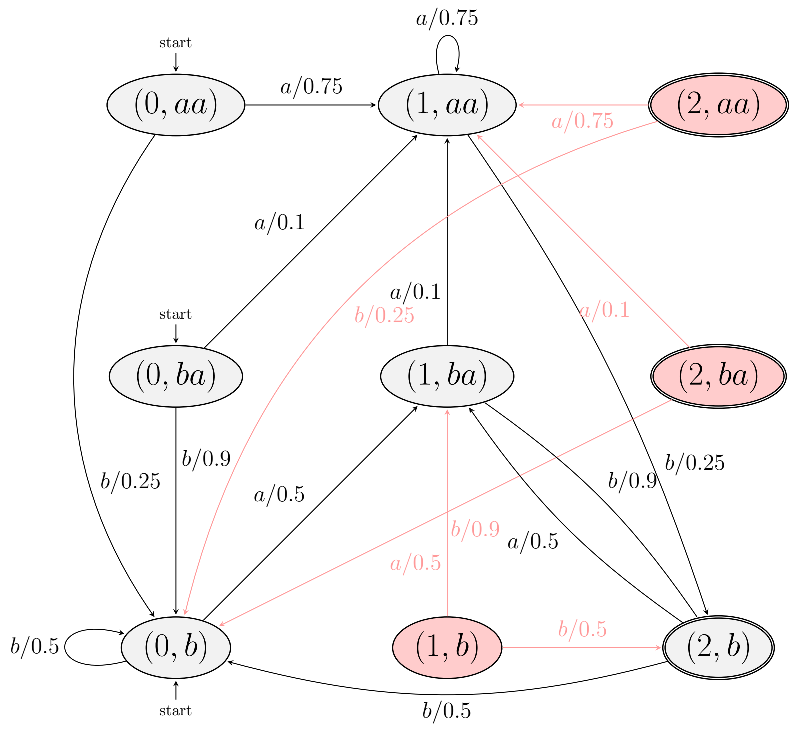
Proving that $\mathscr{L}(M) = \mathscr{L}(M_R)$ is done with induction on the length of strings. The inductive hypothesis is that, for strings $S_{1..k} = t_1 \cdots t_k$ of length k, if $q = (r,s)$ is the state reached by M and q_R the state reached by M_R, then $r = q_R$. Note that both M_R and M are deterministic and complete automata and thus only one state is reached for every string (only one run exists). If a new element t_{k+1} is read, M will move to a new state $q^{'} = (r^{'}, s^{'})$ and M_R to $q_R^{'}$. From the construction of the transitions of M, we see that $r^{'} = q_R^{'}$. Thus, the induction hypothesis holds for $S_{1..k+1}$ as well. It also holds for $k = 0$, since, for every $q = (r,s) \in Q^s$, $r = M_R.q^s$. Therefore, it holds for all k. As a result, if M reaches a final state (r,s), r is reached by M_R. Since $r \in M_R.Q^f$, M_R also reaches a final state.

For proving probabilistic equivalence, first note that the probability of a string given by a predictor P is $P(S_{1..k}) = \prod_{i=1}^{k} P(t_i \mid t_1 \dots t_{i-1})$. Assume now that a *PSA* M_S reads a string $S_{1..k}$ and follows a run $\rho = [l, q_l] \xrightarrow{t_l} [l+1, q_{l+1}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1, q_{k+1}]$. We define a run in a manner similar to that for runs of a *DSFA*. The difference is that a run of a *PSA* may begin at an index $l > 1$, since it may have to wait for l symbols before it can find a state q_l whose label is equal to $S_{1..l}$. We also treat the *PSA* as a reader (not a generator) of strings for which we need to calculate their probability. The probability of $S_{1..k}$ is then given by $P_{M_S}(S_{1..k}) = M_S.\pi(q_l) \cdot \prod_{i=l}^{k} M_S.\gamma(q_i, t_i)$. Similarly, for the embedding M, assume it follows the run $\rho^{'} = [l, q_l^{'}] \xrightarrow{t_l} [l+1, q_{l+1}^{'}] \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_k} [k+1, q_{k+1}^{'}]$. Then, $P_M(S_{1..k}) = M.\pi(q_l^{'}) \cdot \prod_{i=l}^{k} M.\Gamma(q_i^{'}, t_i)$. Now note that M has the same initial state distribution as M_S, i.e., the number of the initial states of M is equal to the number of states of M_S and they have the same distribution. With an inductive proof, as above, we can prove that whenever M reaches a state $q = (r,s)$ and M_S reaches q_S, $s = q_S$. As a result, for the initial states of M and M_S, $M.\pi(q_l^{'}) = M_S.\pi(q_l)$. From the construction of the embedding, we also know that $M_S.\gamma(s, \sigma) = M.\Gamma(q, \sigma)$ for every $\sigma \in \Sigma$. Therefore, $M_S.\gamma(q_i, t_i) = M.\Gamma(q_i^{'}, t_i)$ for every i and $P_M(S_{1..k}) = P_{M_S}(S_{1..k})$. ■

■ **Example 6.15** As an example, consider the *DSFA* M_R of Figure 6.5a for the expression $R = a \cdot b$ with $\Sigma = \{a, b\}$. We present it as a classical automaton, but we remind readers that symbols in Σ correspond to minterms. Figure 6.4a depicts a possible *PST* T that could be learned from a training stream composed of symbols from Σ. Figure 6.4b shows the *PSA* M_S constructed from T. Figure 6.5b shows the embedding M of M_S in M_R that would be created, following the construction procedure of the proof of Theorem 6.2.2. Notice, however, that this embedding has some redundant states and transitions; namely the states indicated with red that have no incoming transitions and are thus inaccessible. The reason is that some states of M_R in Figure 6.5a have a "memory" imbued to them from the structure of the automaton itself. For example, state 2 of M_R has only a single incoming transition with b as its symbol. Therefore, there is no point in merging this state with all the states of M_S, but only with state b. If we follow a straightforward construction, as described above, the result will be the automaton depicted in Figure 6.5b, including the redundant red states. To avoid the inclusion of such states, we can merge M_R and M_S in an incremental fashion (see Algorithm 2). The resulting automaton would then consist only of the black states and transitions of Figure 6.5b. In a streaming setting, we would thus



(a) *DSFA* M_R for $R := a \cdot b$ and $\Sigma = \{a, b\}$.



(b) Embedding of M_S of Figure 6.4b in M_R of Figure 6.5a.

Figure 6.5: Embedding example.

have to wait at the beginning of the stream for some input events to arrive before deciding the start state with which to begin. For example, if b were the first input event, we would then begin with the bottom left state $(0,b)$. On the other hand, if a were the first input event, we would have to wait for yet another event. If another a arrived as the second event, we would begin with the top left state $(0,aa)$. In general, if m is our maximum order, we would need to wait for at most m input events before deciding. ■

Algorithm 2: Embedding of a *PSA* in a *DSFA* (incremental).

```
Input: A DSFA $M_R$ and a PSA $M_S$ learnt with the minterms of $M_R$
Output: An embedding $M$ of $M_S$ in $M_R$ equivalent to both $M_R$ and $M_S$
/* First create the initial states of the merged automaton by combining the
   initial state of $M_R$ with all the states of $M_S$.                        */
1  $Q^s \leftarrow \emptyset$;
2  foreach $s \in M_S.Q$ do
3      $q \leftarrow CreateNewState(M_R.q^s, s)$;
       /* $q$ is a tuple $(r,s)$                                               */
4      $Q^s \leftarrow Q^s \cup \{q\}$;
/* A frontier of states is created, including states of $M$ that have no outgoing
   transitions yet. First frontier consists of the initial states of $M$      */
5  $Frontier \leftarrow Q^s$; $Checked \leftarrow \emptyset$; $\Delta \leftarrow \emptyset$; $\Gamma \leftarrow \emptyset$;
6  while $Frontier \neq \emptyset$ do
7      $q \leftarrow$ pick an element from $Frontier$;
8      foreach $\sigma \in M_S.\Sigma$ do
9          $s^{next} \leftarrow M_S.\tau(q.s,\sigma)$; $r^{next} \leftarrow M_R.\delta(q.r,\sigma)$;
10         if $(r^{next}, s^{next}) \notin Checked$ then
11             $q^{next} \leftarrow CreateNewState(r^{next}, s^{next})$;
12             $Frontier \leftarrow Frontier \cup q^{next}$;
13         else
14             $q^{next} \leftarrow (r^{next}, s^{next})$;
           /* Both the symbol $\sigma$ and its probability are added to the transition.   */
15         $\delta \leftarrow CreateNewTransition(q, \sigma, q^{next})$;
16         $\gamma \leftarrow CreateNewProbability(q, \sigma, M_S.\gamma(q.s,\sigma))$;
17         $\Delta \leftarrow \Delta \cup \delta$; $\Gamma \leftarrow \Gamma \cup \gamma$;
18     $Checked \leftarrow Checked \cup \{q\}$; $Frontier \leftarrow Frontier \setminus \{q\}$;
19 $Q \leftarrow Checked$;
/* Create the final states of $M$ by gathering all states of $M$ whose second element
   is a final state of $M_R$.                                                  */
20 $Q^f \leftarrow \emptyset$;
21 foreach $q \in Q$ do
22     if $q.q_R \in M_R.Q^f$ then
23         $Q^f \leftarrow Q^f \cup \{q\}$;
24 $\Sigma \leftarrow M_S.\Sigma$;
25 return $M = (Q, Q^s, Q^f, \Sigma, \Delta, \Gamma)$;
```

After constructing an embedding M from a *DSFA* M_R and a *PSA* M_S, we can use M to perform forecasting on a test stream. Since M is equivalent to M_R, it can also consume a stream and detect the same instances of the expression R as M_R would detect. However, our goal is to use M to forecast the detection of an instance of R. More precisely, we want to estimate the number of transitions from any state in which M might be until it reaches for the first time one of its final states. Towards this goal, we can use the theory of Markov chains. Let N denote the set of non-final states of M and F the set of its final states. We can organize the transition matrix of M in the following way (we use bold symbols to refer to matrices and vectors and normal ones to refer to scalars or sets):

$$\Pi = \begin{pmatrix} N & N_F \\ F_N & F \end{pmatrix} \tag{6.1}$$

where N is the sub-matrix containing the probabilities of transitions from non-final to non-final states, F the probabilities from final to final states, F_N the probabilities from final to non-final states and N_F the probabilities from non-final to final states. By partitioning the states of a Markov chain into two sets, such as N and F, the following theorem can be used to estimate the probability of reaching a state in F starting from a state in N:

Theorem 6.2.3 — (59). Let Π be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation (6.1) and ξ_{init} its initial state distribution. The probability for the time index n when the system first enters the set of states F, starting from a state in N, can be obtained from

$$P(Y_n \in F, Y_{n-1} \in N, \cdots, Y_2 \in N, Y_1 \in N \mid \xi_{init}) = {\xi_N}^T N^{n-1}(I-N)1 \tag{6.2}$$

where ξ_N is the vector consisting of the elements of ξ_{init} corresponding to the states of N.

In our case, the sets N and F have the meaning of being the non-final and final states of M. The above theorem then gives us the desired probability of reaching a final state.

However, notice that this theorem assumes that we start in a non-final state ($Y_1 \notin F$). A similar result can be given if we assume that we start in a final state.

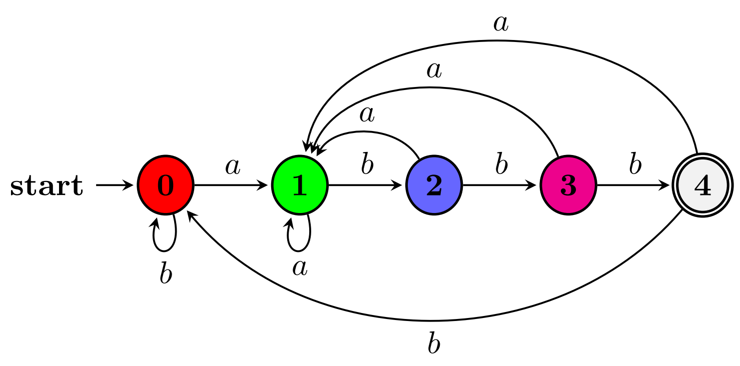
Theorem 6.2.4 Let Π be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation (6.1) and ξ_{init} its initial state distribution. The probability for the time index n when the system first enters the set of states F, starting from a state in F, can be obtained from

$$P(Y_n \in F, Y_{n-1} \in N, \cdots, Y_2 \in N, Y_1 \in F \mid \xi_{init}) = \begin{cases} {\xi_F}^T F 1 & \text{if } n = 2 \\ {\xi_F}^T F_N N^{n-2}(I-N)1 & \text{otherwise} \end{cases} \tag{6.3}$$

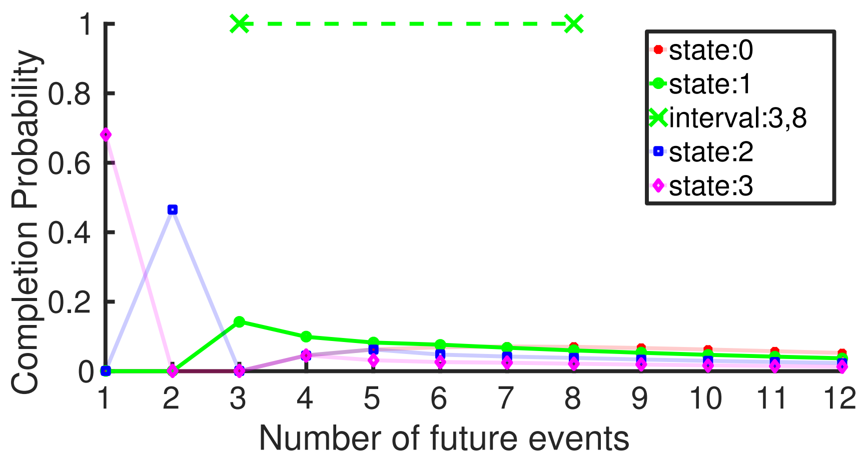
where ξ_F is the vector consisting of the elements of ξ_{init} corresponding to the states of F.

Proof. The proof may be found in the Appendix, Section A.3. ■

Note that the above formulas do not use N_F, as it is not needed when dealing with probability distributions. As the sum of the probabilities is equal to 1, we can derive N_F from N. This is the role of the term $(I-N)1$ in the formulas, which is equal to N_F when there is only a single final state and equal to the sum of the columns of N_F when there are multiple final states, i.e., each element of the matrix corresponds to the probability of reaching one of the final states from a given non-final state.



(a) DFA.



(b) Waiting-time distributions and shortest interval, i.e. $[3,8]$, exceeding a confidence threshold $\theta_{fc} = 50\%$ for state 1.

Figure 6.6: Automaton and waiting-time distributions for $R = a \cdot b \cdot b \cdot b$, $\Sigma = \{a, b\}$.

Using Theorems 6.2.3 and 6.2.4, we can calculate the so-called waiting-time distributions for any state q of the automaton, i.e., the distribution of the index n, given by the waiting-time variable $W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\}$. Theorems 6.2.3 and 6.2.4 provide a way to calculate the probability of reaching a final state, given an initial state distribution ξ_{init}. In our case, as the automaton is moving through its various states, ξ_{init} takes a special form. At any point in time, the automaton is (with certainty) in a specific state q. In that state, ξ_{init} is a vector of 0, except for the element corresponding to the current state of the automaton, which is equal to 1.

■ **Example 6.16** Figure 6.6 shows an example of an automaton (its exact nature is not important, as long as it can also be described as a Markov chain), along with the waiting-time distributions for its non-final states. For this example, if the automaton is in state 2, then the probability of reaching the final state 4 for the first time in 2 transitions is $\approx 50\%$. However, it is 0% for 3 transitions, as the automaton has no path of length 3 from state 2 to state 4. ■

We can use the waiting-time distributions to produce various kinds of forecasts. In the simplest case, we can select the future point with the highest probability and return this point as a forecast. We call this type of forecasting *REGRESSION-ARGMAX*. Alternatively, we may want to know how likely it is that a CE will occur within the next w input events. For this, we can sum the probabilities of the first w points of a distribution and if this

Algorithm 3: Estimating a forecast interval from a waiting-time distribution.

```
Input: A waiting-time distribution $P$ with horizon $h$ and a threshold $\theta_{fc} < 1.0$
Output: The smallest interval $I = (s,e)$ such that $1 \leq s,e \leq h$, $s \leq e$ and $P(I) \geq \theta_{fc}$
1  $s \leftarrow -1$; $e \leftarrow -1$; $i \leftarrow 1$; $j \leftarrow 1$; $p \leftarrow P(1)$;
2  while $j \neq h$ do
       /* Loop invariant: $(s,e)$ is the smallest interval with $P((s,e)) > \theta_{fc}$ among all
          intervals with $e \leq j$ (or $s = e = -1$ in the first iteration).            */
3      /* Expansion phase.                                                                */
4      while $(p < \theta_{fc}) \wedge (j < h)$ do
5          $j \leftarrow j+1$;
6          $p \leftarrow p+P(j)$;
       /* Shrinking phase.                                                                */
7      while $p \geq \theta_{fc}$ do
8          $i \leftarrow i+1$;
9          $p \leftarrow p-P(i)$;
10     $i \leftarrow i-1$;
       /* $s = -1$ indicates that no interval has been found yet, i.e., that this is the
          first iteration.                                                                */
11     if $(spread((i,j)) < spread((s,e))) \vee (s = -1)$ then
12         $s \leftarrow i$;
13         $e \leftarrow j$;
14 return $(s,e)$;
```

sum exceeds a given threshold we emit a "positive" forecast (meaning that a CE is indeed expected to occur); otherwise a "negative" (no CE is expected) forecast is emitted. We call this type of forecasting *CLASSIFICATION-NEXTW*. These kinds of forecasts are easy to compute.

There is another kind of useful forecasts, which are however more computationally demanding. Given that we are in a state q, we may want to forecast whether the automaton, with confidence at least θ_{fc}, will have reached its final state(s) in n transitions, where n belongs to a future interval $I = [start, end]$. The confidence threshold θ_{fc} is a parameter set by the user. The forecasting objective is to select the shortest possible interval I that satisfies θ_{fc}. Figure 6.6b shows the forecast interval produced for state 1 of the automaton of Figure 6.6a, with $\theta_{fc} = 50\%$. We call this third type of forecasting *REGRESSION-INTERVAL*. We have implemented all of the above types of forecasting.

A naive way to estimate the forecast interval from a waiting-time distribution whose domain is $[1,h]$ (we call h, the maximum index of the distribution, its *horizon*) is to first enumerate all possible intervals $(start, end)$, such that $1 \leq start, end \leq h$ and $start \leq end$, and then calculate each interval's probability by summing the probabilities of all of its points. The complexity of such an exhaustive algorithm is $O(h^3)$. To prove this, first note that the algorithm would have to check 1 interval of length h, 2 intervals of length $h-1$, etc., and h intervals of length 1. Assuming that the cost of estimating the probability of an interval is proportional to its length l (l points need to be retrieved and $l-1$ additions be performed), the total cost would thus be:

$$\begin{aligned}
1h + 2(h-1) + 3(h-2) + \cdots + h1 =& \sum_{i=1}^{h} i(h-(i-1)) \\
=& \sum_{i=1}^{h} (ih - i^2 + i) \\
=& h\sum_{i=1}^{h} i - \sum_{i=1}^{h} i^2 + \sum_{i=1}^{h} i \\
=& h\frac{h(h+1)}{2} - \frac{h(h+1)(2h+1)}{6} + \frac{h(h+1)}{2} \\
=& \cdots \\
=& \frac{1}{6}h(h+1)(h+2) \\
=& O(h^3)
\end{aligned}$$

Note that this is just the cost of estimating the probabilities of the intervals, ignoring the costs of actually creating them first and then searching for the best one, after the step of probability estimation.

We can find the best forecast interval with a more efficient algorithm that has a complexity linear in h (see Algorithm 3). We keep two pointers i, j that we initially set them equal to the first index of the distribution. We then repeatedly move i, j in the following manner: We first move j to the right by incrementing it by 1 until $P(i, j)$ exceeds θ_{fc}, where each $P(i, j)$ is estimated incrementally by repeatedly adding $P(j)$ to an accumulator. We then move i to the right by 1 until $P(i, j)$ drops below θ_{fc}, where $P(i, j)$ is estimated by incremental subtractions. If the new interval (i, j) is smaller than the smallest interval exceeding θ_{fc} thus far, we discard the old smallest interval and keep this new one. This wave-like movement of i, j stops when $j = h$. This algorithm is more efficient (linear in the h, see Proposition 6.3.6 in Section 9.4) by avoiding intervals that cannot possibly exceed θ_{fc}. The proof for the algorithm's correctness is presented in the Appendix, Section A.4.

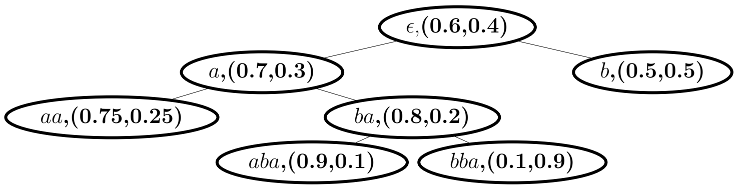
Note that the domain of a waiting-time distribution is not composed of timepoints and thus a forecast does not explicitly refer to time. Each value of the index n on the x axis essentially refers to the number of transitions that the automaton needs to take before reaching a final state, or, equivalently, to the number of future input events to be consumed. If we were required to output forecasts referring to time, we would need to convert these basic event-related forecasts to time-related ones. If input events arrive at regular time intervals, then this conversion is a straightforward multiplication of the forecast by the time interval. However, in the general case where the intervals between input events are not regular and fixed, we would need to build another probabilistic model describing the time that elapses between events and use this model to convert event-related to time-related forecasts. Building such a time model might not always be possible or might be prohibitively expensive. Here we decided to focus on the number of steps for two reasons: a) Sometimes it might not be desirable to give time-related forecasts. Event-related forecasts might be more suitable, as is the case, for example, in the domain of credit card fraud management, where we need to know whether or not the next transaction(s) will be fraudulent. We examine this use case in Section 7.2.3. b) Time-related forecasts might be very difficult (or almost impossible) to produce if the underlying process exhibits a high degree of randomness. For example, this is the case in the maritime domain, where the intervals between vessel position (AIS) messages are wildly random and depend on many (even human-related) factors, e.g., the crew of a vessel simply forgetting to switch on the AIS equipment. In such cases, it might be preferable to perform some form of sampling or interpolation on the original stream of input events in order to derive another stream similar to the original one but with regular intervals. This is the approach we follow in our experiments in the maritime domain (Section 7.2.4). For these reasons, we initially focused on event-related forecasts. This, however, does not exclude the option of using event-related forecasts as a first step in order to subsequently produce time-related ones, whenever this is possible. For example, a simple solution would be to try to model the time elapsed between events via a Poisson process. We intend to pursue this line of work in the future.

Using a Prediction Suffix Tree (*PST*)

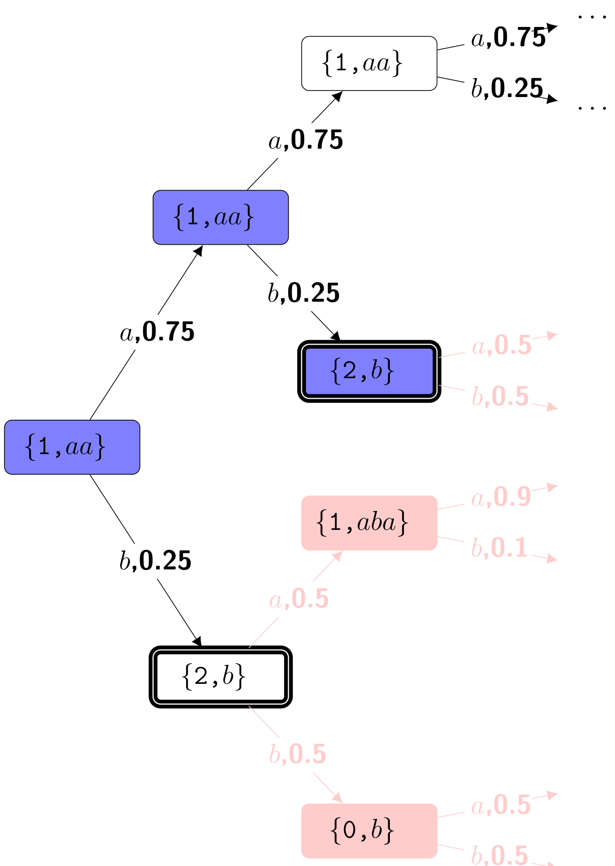
The reason for constructing an embedding of the *PSA* M_S learned from the data into the automaton M_R used for recognition, as described in the previous section, is that the embedding is based on a variable-order model that will consist on average of much fewer states than a full-order model. There is, however, one specific step in the process of creating an embedding that may act as a bottleneck and prevent us from increasing the order to desired values: the step of converting a *PST* to a *PSA*. The number of nodes of a *PST* is often order of magnitudes smaller than the number of states of the *PSA* constructed from that *PST*. Motivated by this observation, we devised a way to estimate the required waiting-time distributions without actually constructing the embedding. Instead, we make direct use of the *PST*, which is more memory efficient. Thus, given a *DSFA* M_R and its *PST* T, we can estimate the probability for M_R to reach for the first time one of its final states in the following manner.

As the system processes events from the input stream, besides feeding them to M_R, 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. The probability of any future sequence of events can be estimated with the use of the probability distribution at l. In other words, if $S_{1..k} = \cdots, t_{k-1}, t_k$ is the stream seen thus far, then the next symbol probability for t_{k+1}, i.e., $P(t_{k+1} \mid t_{k-m+1}, \cdots, t_k)$, can be directly retrieved from the distribution of the leaf l. If we want to look further into the future, e.g., into t_{k+2}, we can repeat the same process as necessary. Namely, if we fix t_{k+1}, then the probability for t_{k+2}, $P(t_{k+2} \mid t_{k-m+2}, \cdots, t_{k+1})$, can be retrieved from T, by retrieving the leaf $l^{'}$ reached with $t_{k+1}, \cdots, t_{k-m+2}$. In this manner, we can estimate the probability of any future sequence of events. Consequently, we can also estimate the probability of any future sequence of states of the *DSFA* M_R, since we can simply feed these future event sequences to M_R and let it perform "forward" recognition with these projected events. In other words, we can let M_R "generate" a sequence of future states, based on the sequence of projected events, in order to determine when M_R will reach a final state. Finally, since we can estimate the probability for any future sequence of states of M_R, we can use the definition of the waiting-time variable ($W_q = inf\{n : Y_0, Y_1, ..., Y_n, Y_0 = q, q \in Q \backslash F, Y_n \in F\}$) to calculate the waiting-time distributions.

■ **Example 6.17** Figure 6.7 shows an example of this process for the automaton M_R of Figure 6.5a. Figure 6.7a displays an example *PST* T learned with the minterms/symbols of M_R. ■



(a) The *PST* T for the automaton M_R of Figure 6.5a.



(b) Future paths followed by M_R and T starting from state 1 of M_R 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 M_R.

Figure 6.7: Example of estimating a waiting-time distribution without a Markov chain.

R One remark should be made at this point in order to showcase how an attempt to convert T to a PSA could lead to a blow-up in the number of states. The basic step in such a conversion is to take the leaves of T and use them as states for the PSA. If this was sufficient, the resulting PSA would always have fewer states than the PST. As this example shows, this is not the case. Imagine that the states of the PSA are just the leaves of T and that we are in the right-most state/node, $b,(0.5,0.5)$. What will happen if an a event arrives? We would be unable to find a proper next state. The state $aa,(0.75,0.25)$ is obviously not the correct one, whereas states $aba,(0.9,0.1)$ and $bba,(0.1,0.9)$ are both "correct", in the sense that ba is a suffix of both aba and bba. In order to overcome this ambiguity regarding the correct next state, we would have to first expand node $b,(0.5,0.5)$ of T and then use the children of this node as states of the PSA. In this simple example, this expansion of a single problematic node would not have serious consequences. But for deep trees and large alphabets, the number of states generated by such expansions are far more than the number of the original leaves. For this reason, the size of the PSA is far greater than that of the original, unexpanded PST.

■ **Example 6.18** Figure 6.7b illustrates how we can estimate the probability for any future sequence of states of the $DSFA$ M_R, using the distributions of the PST T. Let us assume that, after consuming the last event, M_R is in state 1 and T has reached its left-most node, $aa,(0.75,0.25)$. This is shown as the left-most node also in Figure 6.7b. Each node in this figure has two elements: the first one is the state of M_R 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 6.7b, 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 M_R will move to state 2 and T will reach leaf $b,(0.5,0.5)$. This is shown in Figure 6.7b as the white node $\{2,b\}$. This node has a double border to indicate that M_R 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 $\{1,aa\}$. 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 6.7b, 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 on 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. ■

Note that the tree-like structure of Figure 6.7b is not an actual data structure that we need to construct and maintain. It is only a graphical illustration of the required computation steps. The actual computation is performed recursively on demand. At each recursive call, a new frontier of virtual future nodes at level k is generated. We thus do not maintain all the nodes of this tree in memory, but only access the PST T, which is typically much more compact than a PSA. Despite this fact though, the size of the frontier after each recursive call grows exponentially as we try to look deeper into the future. This cost can be significantly reduced by employing the following optimizations. First, note in Figure 6.7b, that the paths starting from the two $\{2,b\}$ nodes are pink. This indicates that these paths do not actually need to be generated, as they start from a final state. We are only interested in the first time M_R reaches a final state and not in the second, third, etc. As a result, paths with more than one final states are not useful. With this optimization, we can still do an exact estimation of the waiting-time distribution. Another useful optimization is to prune paths that we know will have a very low probability, even if they are necessary for an exact estimation of the distributions. The intuition is that such paths will not contribute significantly to the probabilities of our waiting-time distribution, even if we do expand them. We can prune such paths, accepting the risk that we will have an approximate estimation of the waiting-time distribution. This pruning can be done without generating the paths in their totality. As soon as a partial path has a low probability, we can stop its expansion, since any deeper paths will have even lower probabilities. We have found this optimization to be very efficient while having negligible impact on the distribution for a wide range of cut-off thresholds.

6.2.5 Estimation of empirical probabilities

We have thus far described how an embedding of a *PSA* M_S in a *DSFA* M_R can be constructed and how we can estimate the forecasts for this embedding. We have also presented how this can be done directly via a *PST*, without going through a *PSA*. However, before learning the *PST*, as described in Section 6.2.3, we first need to estimate the empirical probabilities for the various symbols. We describe here this extra initial step. In [121], it is assumed that, before learning a *PST*, the empirical probabilities of symbols given various contexts are available. The suggestion in [121] is that these empirical probabilities can be calculated either by repeatedly scanning the training stream or by using a more time-efficient algorithm that keeps pointers to all occurrences of a given context in the stream. We opt for a variant of the latter choice.

First, note that the empirical probabilities of the strings (s) and the expected next symbols (σ) observed in a stream are given by the following formulas [121]:

$$\hat{P}(s) = \frac{1}{k-m} \sum_{j=m}^{k-1} \chi_j(s) \tag{6.4}$$

$$\hat{P}(\sigma \mid s) = \frac{\sum_{j=m}^{k-1} \chi_{j+1}(s \cdot \sigma)}{\sum_{j=m}^{k-1} \chi_j(s)} \tag{6.5}$$

where k is the length of the training stream $S_{1..k}$, m is the maximum length of the strings (s) that will be considered and

$$\chi_j(s) = \begin{cases} 1 & \text{if } S_{(j-|s|+1)\cdots j} = s \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

In other words, we need to count the number of occurrences of the various candidate strings s in $S_{1..k}$. The numerators and denominators in Eq. (6.4) and (6.5) are essentially counters for the various strings.

In order to keep track of these counters, we can use a tree data structure which allows to scan the training stream only once. We call this structure a *Counter Suffix Tree* (*CST*). Each node in a *CST* is a tuple (σ, c) where σ is a symbol from the alphabet (or ε only for the root node) and c a counter. For each level k of the tree, it always holds that $SumOfCountersAtK \leq ParentCounter$ and $SumOfCountersAtK \geq ParentCounter - (k-1)$. By following a path from the root to a node, we get a string $s = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$, where $\sigma_0 = \varepsilon$ corresponds to the root node. The property maintained as a *CST* is built from a stream $S_{1..k}$ is that the counter of the node σ_n that is reached with s gives us the number of occurrences of the string $\sigma_n \cdot \sigma_{n-1} \cdots \sigma_1$ (the reversed version of s) in $S_{1..k}$.

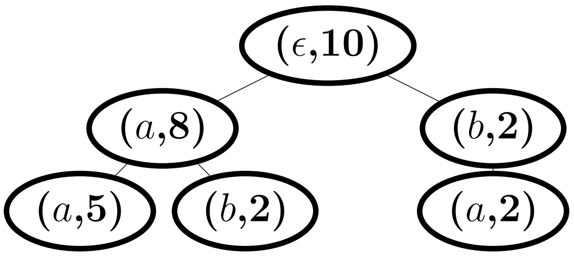


Figure 6.8: Example of a Counter Suffix Tree with $m = 2$ and $S = aaabaabaaa$.

■ **Example 6.19** As an example, see Figure 6.8, which depicts the *CST* of maximum depth 2 for the stream $S = aaabaabaaa$. If we want to retrieve the number of occurrences of the string $b \cdot a$ in S, we follow the left child $(a, 7)$ of the root and then the right child of this. We thus reach $(b, 2)$ and indeed $b \cdot a$ occurs twice in S. ■

A *CST* can be incrementally constructed by maintaining a buffer of size m that always holds the last m symbols of S. The contents of the buffer are fed into the *CST* after the arrival of a new symbol. The update algorithm follows a path through the *CST* according to the whole string provided by the buffer. For every node that already exists, its counter is incremented by 1. If a node does not exist, it is created and its counter is set to 1. At any point, having been updated with the training stream, the *CST* can be used to retrieve the necessary counters and estimate the empirical probabilities of Equations (6.4) and (6.5) that are subsequently used in the *PST* construction.

6.3 Complexity analysis

Figure 6.9 depicts the steps required for estimating forecasts, along with the input required for each of them. The first step (box 1) takes as input the minterms of a *DSFA*, the maximum order m of dependencies to be captured and a training stream. Its output is a *CST* of maximum depth m (Section 6.2.5). In the next step (box 2), the *CST* is converted to a *PST*, using an approximation parameter α and a parameter n for the maximum number of states for the *PSA* to be constructed subsequently (Section 6.2.3). For the third step, we have two options: we can either use the *PST* to directly estimate the waiting-time distributions (box 3b, Section 14) or we can convert the *PST* to a *PSA*, by using the leaves of the *PST* as states of the *PSA* (box 3a, Section 6.2.3). If we follow the first path, we can then move on directly to the last step of estimating the actual forecasts, using the confidence threshold θ_{fc} provided by the user (box 6). If we follow the alternative path, the *PSA* is merged with the initial *DSFA* to create the embedding of the *PSA* in the *DSFA* (box 4, Section 6.2.4). From the embedding we can calculate the waiting-time distributions (box 5), which can be used to derive the forecasts (box 6).

The learning algorithm of step 2, as presented in [121], is polynomial in m, n, $\frac{1}{\alpha}$ and the size of the alphabet (number of minterms in our case). Below, we give complexity results for the remaining steps.

Proposition 6.3.1 — Step 1 in Figure 6.9. Let $S_{1..k}$ be a stream and m the maximum depth of the Counter Suffix Tree T to be constructed from $S_{1..k}$. The complexity of constructing T is $O(m(k-m))$.

Proof. See Appendix, Section A.5.1. ■

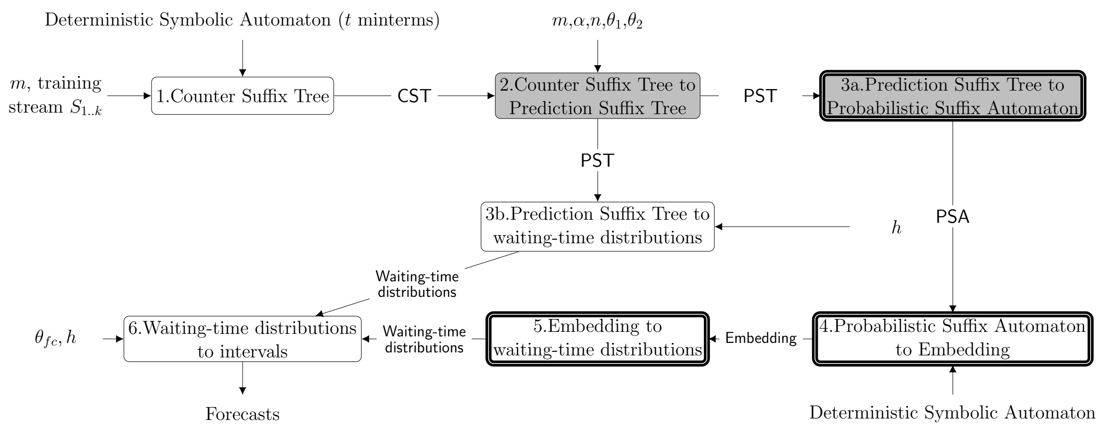

Deterministic Symbolic Automaton (t minterms)
$m,\alpha,n,\theta_1,\theta_2$
m, training stream $S_{1..k}$
1.Counter Suffix Tree
CST
2.Counter Suffix Tree to Prediction Suffix Tree
PST
3a.Prediction Suffix Tree to Probabilistic Suffix Automaton
PST
3b.Prediction Suffix Tree to waiting-time distributions
h
PSA
Waiting-time distributions
θ_{fc}, h
6.Waiting-time distributions to intervals
Waiting-time distributions
5.Embedding to waiting-time distributions
Embedding
4.Probabilistic Suffix Automaton to Embedding
Forecasts
Deterministic Symbolic Automaton

Figure 6.9: Steps for calculating forecasts. $S_{1..k}$: training stream, m: maximum assumed order, α: approximation parameter, n: maximum number of states for the *PSA*, θ_{fc}: confidence threshold, θ_1, θ_2: thresholds for learning a *PST*, k: size of training stream, t: number of minterms, w: forecasting window. Gray blocks indicate steps described in [121]. White blocks indicate steps introduced in this thesis. Blocks with double borders represent the steps required exclusively when going through a PSA embedding, while single-border blocks represent steps bypassing PSA construction.

Proposition 6.3.2 — Step 3a in Figure 6.9. Let T be a *PST* of maximum depth m, learned with the t minterms of a *DSFA* M_R. The complexity of constructing a *PSA* M_S from T is $O(t^{m+1} \cdot m)$.

Proof. See Appendix, Section A.5.2. ■

Proposition 6.3.3 — Step 3b in Figure 6.9. Let T be a *PST* of maximum depth m, learned with the t minterms of a *DSFA* M_R. The complexity of estimating the waiting-time distribution for a state of M_R and a horizon of length h directly from T is $O((m+3)\frac{t-t^{h+1}}{1-t})$.

Proof. See Appendix, Section A.5.6. ■

Proposition 6.3.4 — Step 4 in Figure 6.9. Let M_R be a *DSFA* with t minterms and M_S a *PSA* learned with the minterms of M_R. The complexity of constructing an embedding M of M_S in M_S with Algorithm 2 is $O(t \cdot |M_R.Q \times M_S.Q|)$.

Proof. See Appendix, Section A.5.3. ■

Notice that the cost of learning a *PSA* might be exponential in m. In the worst case, all permutations of the t minterms of length m need to be added to the *PST* and the *PSA*. This may happen if the statistical properties of the training stream are such that all these permutations are deemed as important. In this case, the final embedding in the *DSFA* M_R might have up to $t^m \cdot |M_R.Q|$ states. This is also the upper bound of the number of states of the automaton the would be created using the method of [6], where every state of an initial automaton is split into at most t^m sub-states, regardless of the properties of the stream. Thus, in the worst case, our approach would create an automaton of size similar to an automaton encoding a full-order Markov chain. Our approach provides an advantage when the statistical properties of the training stream allow us to retain only some of the dependencies of order up to m.

Proposition 6.3.5 — Step 5 in Figure 6.9. Let M be the embedding of a *PSA* M_S in a *DSFA* M_R. The complexity of estimating the waiting-time distribution for a state of M and a horizon of length h using Theorem 6.2.3 is $O((h-1)k^{2.37})$, where k is the dimension of the square matrix N. A similar result may be obtained for Theorem 6.2.4.

Proof. See Appendix, Section A.5.4. ■

Proposition 6.3.6 — Step 6 in Figure 6.9. For a waiting-time distribution with a horizon of length h, the complexity of finding the smallest interval that exceeds a confidence threshold θ_{fc} with Algorithm 3 is $O(h)$.

Proof. See Appendix, Section A.5.5. ■

The complexity of the last step (6), when the forecasts are "classification" decisions about whether a CE will occur within the next w input events, is $O(w)$. In order to make such a positive or negative decision, we can simply sum the probabilities of the first w points of a waiting-time distribution and compare this sum to the given threshold θ_{fc}. If this sum exceeds the given threshold, then the decision is positive. The cost of the summation is $O(w)$.

7. Forecasting with suffix trees: experiments

In this chapter we present empirical results with prediction suffix trees. Before doing so, we first start with a discussion of an important topic which has not received much attention thus far in our thesis: the topic of precisely how forecasts can and should be evaluated in a CEF task. We thus first discuss how we can quantify the quality of forecasts in Section 7.1. We finally demonstrate the efficacy of our framework in Section 7.2, by showing experimental results on two application domains. We conclude with Section 7.3 with a brief summary of our work.

7.1 Measuring the quality of forecasts

As described in Section 6.2.4, there are various types of forecasts that could be produced from the waiting-time distributions of an automaton. In this section, we discuss in more detail these different forecasting tasks and how the quality of the produced forecasts can be quantified in each case. We distinguish three different types of forecasting tasks: a) SDE forecasting, where our goal is to forecast the next SDE in the input stream; b) regression CE forecasting, where our goal is to forecast when a CE will occur (either *REGRESSION-ARGMAX* or *REGRESSION-INTERVAL*); c) classification CE forecasting, where our goal is to forecast whether or not a CE will occur within a short future window (*CLASSIFICATION-NEXTW*).

7.1.1 SDE forecasting

Although our main focus is on forecasting occurrences of CEs, we can start with a simpler task, targeting SDEs. Not only does this allow us to establish a baseline with some more easily interpretable results, but it also enables us to show the differences between SDE and CE forecasting. As we will show, CE forecasting is more challenging than SDE forecasting, in terms of the feasibility of looking deep into the past. Another reason for running experiments for SDE forecasting is to find the best values for the hyperparameters used for learning a prediction suffix tree. Since it is much faster to run this type of experiments, compared to CE forecasting experiments, we can use a hypergrid of hyperparameter values and for each hypergrid point we run SDE forecasting.

In this type of experiments, our goal is to investigate how well our proposed framework can forecast the next SDE to appear in the stream, given that we know the last m SDEs. This task is the equivalent of *next symbol prediction* in the terminology of the compression community [24]. As explained in Section 6.2.2, the metric that we use to estimate the quality of a predictor $\hat{P}$ is the average log-loss with respect to a test sequence $S_{1..k} = t_1, t_2, \cdots, t_k$, given by $l(\hat{P}, S_{1..k}) = -\frac{1}{T}\sum_{i=1}^{k} log\hat{P}(t_i \mid t_1 \cdots t_{i-1})$. The lower the average log-loss, the better the predictor is assumed to be.

We remind that the "symbols" which we try to predict in these experiments are essentially the minterms of our *DSFA* in our case. In other words, we do not try to predict exactly what the next SDE will be, but which minterm the next SDE will satisfy. For example, if we have the minterms of Table 6.3, then our task is to predict whether the next SDE will satisfy ψ_A (i.e., the speed of the vessel will be below 5 knots), ψ_B (i.e., the speed will be above 20 knots) or $\neg\psi_A \wedge \neg\psi_B$ (i.e., the speed will be between 5 and 20 knots).

7.1.2 Regression CE forecasting

After SDE forecasting, we move on to regression CE forecasting. Our goal in this task to forecast when a CE will occur. We call them *regression* experiments due to the fact that the forecasts are "continuous" values, in the form of forecast intervals/points. This is in contrast to the next forecasting task, where each forecast is a binary value, indicating whether a CE will occur or not and is thus called a *classification* task.

One important difference between SDE and CE forecasting (both regression and classification) is that, in SDE forecasting, a forecast is emitted after every new SDE is consumed. On the other hand, in CE forecasting, emitting a forecast after every new SDE is feasible in principle, but not very useful and can also produce results that are misleading. By their very nature, CEs are relatively rare within a stream of input SDEs. As a result, if we emit a forecast after every new SDE, some of these forecasts (possibly even the vast majority) will have a significant temporal distance from the CE to which they refer. As an example, consider a pattern from the maritime domain which detects the entrance of a vessel in the port of Tangiers. We can also try to use this pattern for forecasting, with the goal of predicting when the vessel will arrive at the port of Tangiers. However, the majority of the vessel's messages may lie in areas so distant from the port (e.g., in the Pacific ocean) that it would be practically useless to emit forecasts when the vessel is in these areas. Moreover, if we do emit forecasts from these distant areas, the scores and metrics that we use to evaluate the quality of the forecasts will be dominated by these, necessarily low-quality, distant forecasts.

For these reasons, before running a regression experiment, we must first go through a preprocessing step. We must find the timepoints within a stream where it is "meaningful" to emit forecasts. We call these timepoints the *checkpoints* of the stream. To do so, we must first perform recognition on the stream to find the timepoints where CEs are detected. We then set a required distance d that we want to separate a forecast from its CE, in the sense that we require each forecast to be emitted d events before the CE. After finding all the CEs in a stream and setting a value for d, we set as checkpoints all the SDEs which occur d events before the CEs. This typically means that we end up with as many checkpoints as CEs for a given value of d (unless the distance between two consecutive CEs is smaller than d, in which case no checkpoint for the second CE exists). We can then show results for various values of d, starting from the smallest possible value of 1 (i.e., emitting forecasts from the immediately previous SDE before the CE).

At each checkpoint, a forecast interval is produced, as per Section 6.2.4. Some of the metrics we can use to assess the quality of the forecasts assume that forecasts are in the form of points. Such point metrics are the following: the Root Mean Squared Error (RMSE) and the Mean Absolute Error (MAE) (the latter is less sensitive than RMSE to outliers). If we denote by C the set of all checkpoints, by y_c the actual distance (in number of events) between a checkpoint c and its CE (which is always d) and by $\hat{y}_c$ our forecast, then the definitions for RMSE and MAE are as follows:

$$RMSE = \sqrt{\frac{1}{|C|}\sum_{c \in C} |\hat{y}_c - y_c|^2} \tag{7.1}$$

$$MAE = \frac{1}{|C|}\sum_{c \in C} |\hat{y}_c - y_c| \tag{7.2}$$

When these metrics are used, we need to impose an extra constraint on the forecasts, requiring that the maximum spread of each forecast is 0, i.e., we produce point (instead of interval) forecasts.

Besides these points metrics, we can also use an interval metric, the so-called *negatively oriented interval score* [65]. If $\hat{y}_c = (l_c, u_c)$ is an interval forecast produced with confidence $b = 1 - a$ and y_c the actual observation (distance), then the negatively oriented interval score (NOIS) for this forecast is given by:

$$NOIS_c = (u_c - l_c) + \frac{2}{a}(l_c - y_c)I_{x<l_c} + \frac{2}{a}(y_c - u_c)I_{x>u_c} \tag{7.3}$$

We can then estimate the average negatively oriented interval score (ANOIS) as follows:

$$ANOIS = \frac{1}{|C|}\sum_{c \in C} NOIS_c \tag{7.4}$$

The best possible value for ANOIS is 0 and is achieved only when all forecasts are point forecasts (so that $u_c - l_c$ is always 0) and all of them are also correct (so that the last two terms in Eq. (7.3) are always 0). In every other case, a forecast is penalized if its interval is long (so that focused intervals are promoted), regardless of whether it is correct. If it is indeed correct, no other penalty is added. If it is not correct, then an extra penalty is added, which is essentially the deviation of the forecast from the actual observation, weighted by a factor that grows with the confidence of the forecast. For example, if the confidence is 100%, then $b = 1.0$, $a = 0.0$ and the extra penalty, according to Eq. (7.3), grows to infinity. Incorrect forecasts produced with high confidence are thus severely penalized. Note that if we emit only point forecasts ($\hat{y}_c = l_c = u_c$), then NOIS and ANOIS could be written as follows:

$$NOIS_c = \frac{2}{a}|\hat{y}_c - y_c| \tag{7.5}$$

$$ANOIS = \frac{1}{|C|}\sum_{c \in C} \frac{2}{a}|\hat{y}_c - y_c| \tag{7.6}$$

ANOIS then becomes a weighted version of MAE.

7.1.3 Classification CE forecasting

The last forecasting task is the most challenging. In contrast to regression experiments, where we emit forecasts only at a specified distance before each CE, in classification experiments we emit forecasts regardless of whether a CE occurs or not. The goal is to predict the occurrence of CEs within a short future window or provide a "negative" forecast if our model predicts that no CE is likely to occur within this window. In practice, this task could be the first one performed at every new event arrival. If a positive forecast is emitted, then the regression task could also be performed in order to pinpoint more accurately when the CE will occur.

One issue with classification experiments is that it is not so straightforward to establish checkpoints in the stream. In regression experiments, CEs provide a natural point of reference. In classification experiments, we do not have such reference points, since we are also required to predict the absence of CEs. As a result, instead of using the stream to find checkpoints, we can use the structure of the automaton itself. We may not know the actual distance to a CE, but the automaton can provide us with an "expected" or "possible" distance, as follows. For an automaton that is in a final state, it can be said that the distance to a CE is 0. More conveniently, we can say that the "process" which it describes has been completed or, equivalently, that there remains 0% of the process until completion. For an automaton that is in a non-final state but separated from a final state by 1 transition, it can be said that the "expected" distance is 1. We use the term "expected" because we are not interested in whether the automaton will actually take the transition to a final state. We want to establish checkpoints both for the presence and the absence of CEs. When the automaton fails to take the transition to a final state (and we thus have an absence of a CE), this "expected" distance is not an actual distance, but a "possible" one that failed to materialize. We also note that there might also exist other walks from this non-final state to a final one whose length could be greater than 1 (in fact, there might exist walks with "infinite length", in case of loops). In order to estimate the "expected" distance of a non-final state, we only use the shortest walk to a final state.

After estimating the expected distances of all states, we can then express them as percentages by dividing them by the greatest among them. A 0% distance will thus refer to final states, whereas a 100% distance to the state(s) that are the most distant to a final state, i.e., the automaton has to take the most transitions to reach a final state. These are the start states. We can then determine our checkpoints by specifying the states in which the automaton is permitted to emit forecasts, according to their "expected" distance. For example, we may establish checkpoints by allowing only states with a distance between 40% and 60% to emit forecasts. The intuition here is that, by increasing the allowed distance, we make the forecasting task more difficult. Another option for measuring the distance of a state to a possible future CE would be to use the waiting-time distribution of the state and set its expectation as the distance. However, this assumes that we have gone through the training phase first and learned the distributions. For this reason, we avoid using this way to estimate distances.

The evaluation task itself consists of the following steps. At the arrival of every new input event, we first check whether the distance of the new automaton state falls within the range of allowed distances, as explained above. If the new state is allowed to emit a forecast, we use its waiting-time distribution to produce the forecast. Two parameters are taken into account: the length of the future window w within which we want to know whether a CE will occur and the confidence threshold θ_{fc}. If the probability of the first w points of the distribution exceeds the threshold θ_{fc}, we emit a positive forecast, essentially affirming that a CE will occur within the next w events; otherwise, we emit a negative forecast, essentially rejecting the hypothesis that a CE will occur. We thus obtain a binary classification task.

As a consequence, we can make use of standard classification measures, like precision and recall. Each forecast is evaluated: a) as a *true positive* (TP) if the forecast is positive and the CE does indeed occur within the next w events from the forecast; b) as a *false positive* (FP) if the forecast is positive and the CE does not occur; c) as a *true negative* (TN) if the forecast is negative and the CE does not occur and d) as a *false negative* (FN) if the forecast is negative and the CE does occur; Precision is then defined as $Precision = \frac{TP}{TP+FP}$ and recall (also called sensitivity or true positive rate) as $Recall = \frac{TP}{TP+FN}$. As already mentioned, CEs are relatively rare in a stream. It is thus important for a forecasting engine to be as specific as possible in identifying the true negatives. For this reason, besides precision and recall, we also use *specificity* (also called true negative rate), defined as $Specificity = \frac{TN}{TN+FP}$.

A classification experiment is performed as follows. For various values of the "expected" distance and the confidence threshold θ_{fc}, we estimate precision, recall and specificity on a test dataset. For a given distance, θ_{fc} acts as a cut-off parameter. For each value of θ_{fc}, we estimate the recall (sensitivity) and specificity scores and we plot these scores as a ROC curve. For each distance, we then estimate the area under curve (AUC) for the ROC curves. The higher the AUC value, the better the model is assumed to be.

The setting described above is the most suitable for evaluation purposes, but might not be the most appropriate when such a system is actually deployed. For deployment purposes, another option would be to simply set a best, fixed confidence threshold (e.g., by selecting, after evaluation, the threshold with the highest F1-score or Matthews correlation coefficient) and emit only positive forecasts, regardless of their distance. Forecasts with low probabilities (i.e., negative forecasts) will thus be ignored/suppressed. This is justified by the fact that a user would typically be more interested in positive forecasts. For evaluation purposes, this would not be an appropriate experimental setting, but it would suffice for deployment purposes, where we would then be focused on fine-tuning the confidence threshold. Here we focus on evaluating our system and thus do not discuss further any deployment solution.

7.2 Empirical evaluation

We now present experimental results on two datasets, a synthetic one (Section 7.2.3) and a real-world one (Section 7.2.4). We first briefly discuss in Section 7.2.1 the models that we evaluated and present our software and hardware settings in Section 7.2.2. We show results for SDE forecasting and classification CE forecasting. Due to space limitations, we omit results for regression CE forecasting . We intend to present them in future work.

7.2.1 Models tested

In the experiments that we present, we evaluated the variable-order Markov model that we have presented in the previous chapter in its two versions: the memory efficient one that bypasses the construction of a Markov chain and makes direct use of the *PST* learned from a stream (Section 14) and the computationally efficient one that constructs a *PSA* (Section 6.2.4). We compared these against four other models inspired by the relevant literature.

The first, described in [6, 8], is the most similar in its general outline to our proposed method. It is a previous version of our system presented in this and the previous chapters and is also based on automata and Markov chains. The main difference is that it attempts to construct full-order Markov models of order m and is thus typically restricted to low values for m.

The second model is presented in [101], where automata and Markov chains are used once again. However, the automata are directly mapped to Markov chains and no attempt is made to ensure that the Markov chain is of a certain order. Thus, in the worst case, this model essentially makes the assumption that SDEs are i.i.d. and $m = 0$.

As a third alternative, we evaluated a model that is based on Hidden Markov Models (HMM), similar to the work presented in [110]. That work uses the Esper event processing engine [54] and attempts to model a business process as a HMM. For our purposes, we use a HMM to describe the behavior of an automaton, constructed from a given symbolic regular expression. The observation variable of the HMM corresponds to the states of the automaton. Thus, the set of possible values of the observation variable is the set of automaton states. An observation sequence of length l for the HMM consists of the sequence of l states visited by the automaton after consuming l SDEs. The l SDEs (symbols) are used as values for the hidden variable. The last l symbols are the last l values of the hidden variable. Therefore, this HMM always has l hidden states, whose values are taken from the SDEs, connected to l observations, whose values are taken from the automaton states. We can train such a HMM with the Baum-Welch algorithm, using the automaton to generate a training observation sequence from the original training stream. We can then use this learned HMM to produce forecasts on a test dataset. We produce forecasts in an online manner as follows: as the stream is consumed, we use a buffer to store the last l states visited by the pattern automaton. After every new event, we "unroll" the HMM using the contents of the buffer as the observation sequence and the transition and emission matrices learned during the training phase. We can then use the forward algorithm to estimate the probability of all possible future observation sequences (up to some length), which, in our case, correspond to future states visited by the automaton. Knowing the probability of every future sequence of states allows us to estimate the waiting-time distribution for the current state of the automaton and thus build a forecast, as already described. Note that, contrary to the previous approaches, the estimation of the waiting-time distribution via a HMM must be performed online. We cannot pre-compute the waiting-time distributions and store the forecasts in a look-up table, due to the possibly large number of entries. For example, assume that $l = 5$ and the size of the "alphabet" (SDE symbols) of our automaton is 10. For each state of the automaton, we would have to pre-compute 10^5 entries. In other words, as with Markov chains, we still have a problem of combinatorial explosion. We try to "avoid" this problem by estimating the waiting-time distributions online.

Our last model is inspired by the work presented in [1]. This method comes from the process mining community and has not been previously applied to CEF. However, due to its simplicity, we use it here as a baseline method. We again use a training dataset to learn the model. In the training phase, every time the pattern automaton reaches a certain state q, we simply count how long (how many transitions) we have to wait until it reaches a final state. After the training dataset has been consumed, we end up with a set of such "waiting times" for every state. The forecast to be produced by each state is then estimated simply by calculating the average "waiting time".

As far as the Markov models are concerned, we try to increase their order to the highest possible value, in order to determine if and how high-order values offer an advantage. We have empirically discovered that our system can efficiently handle automata and Markov chains that have up to about 1200 states. Beyond this point, it becomes almost prohibitive (with our hardware) to create and handle transition matrices with more than 1200^2 elements. We have thus set this number as an upper bound and increased the order of a model until this number is reached. This restriction is applied both to full-order models and variable-order models that use a *PSA* and an embedding, since in both of these cases we need to construct a Markov chain. For the variable-order models that make direct use of a *PST*, no Markov chain is constructed. We thus increase their order until their performance scores seem to reach a stable number or a very high number, beyond which it makes little sense to continue testing.

7.2.2 Hardware and software settings

All experiments were run on a 64-bit Debian 10 (buster) machine with Intel Core i7-8700 CPU @ 3.20GHz x 12 processors and 16 GB of memory. Our framework was implemented in Scala 2.12.10. We used Java 1.8, with the default values for the heap size. For the HMM models, we relied on the Smile machine learning library [129]. All other models were developed by us. No attempt at parallelization was made.

7.2.3 Credit card fraud management

The first dataset used in our experiments is a synthetic one, inspired by the domain of credit card fraud management [21]. We start with a synthetically generated dataset in order to investigate how our method performs under conditions that are controlled and produce results more readily interpretable. The data generator was developed in collaboration with Feedzai[1], our partner in the SPEEDD project[2].

In this dataset, each event is supposed to be a credit card transaction, accompanied by several arguments, such as the time of the transaction, the card ID, the amount of money spent, the country where the transaction took place, etc. In the real world, a very small proportion of such transactions are fraudulent and the goal of a CER system would be to detect, with very low latency, fraud instances. To do so, a set of fraud patterns must be provided to the engine. For typical cases of such patterns in a simplified form, see [21]. In our experiments, we use one such pattern, consisting of a sequence of consecutive transactions, where the amount spent at each transaction is greater than that of the previous transaction. Such a trend of steadily increasing amounts constitutes a typical fraud pattern. The goal in our forecasting experiments is to predict if and when such a pattern will be completed, even before it is detected by the engine (if in fact a fraud instance occurs), so as to possibly provide a wider margin for action to an analyst.

We generated a dataset consisting of 1,000,000 transactions in total from 100 different cards. About 20% of the transactions are fraudulent. Not all of these instances of fraud belong to the pattern of increasing amounts. We actually inject seven different types of known fraudulent patterns in the dataset, including, for instance, a decreasing trend. Each fraudulent sequence for the increasing trend consists of eight consecutive transactions with increasing amounts, where the amount is increased each time by 100 monetary units or more. We additionally inject sequences of transactions with increasing amounts, which are not fraudulent. In those cases, we randomly interrupt the sequence before it reaches the eighth transaction. In the legitimate sequences the amount is increased each time by 0 or more units. With this setting, we want to test the effect of long-term dependencies on

[1]`https://feedzai.com`
[2]`http://speedd-project.eu`

the quality of the forecasts. For example, a sequence of six transactions with increasing amounts, where all increases are 100 or more units is very likely to lead to a fraud detection. On the other hand, a sequence of just two transactions with the same characteristics, could still possibly lead to a detection, but with a significantly reduced probability. We thus expect that models with deeper memories will perform better. We used 75% of the dataset for training and the rest for testing. No k-fold cross validation is performed, since each fold would have exactly the same statistical properties.

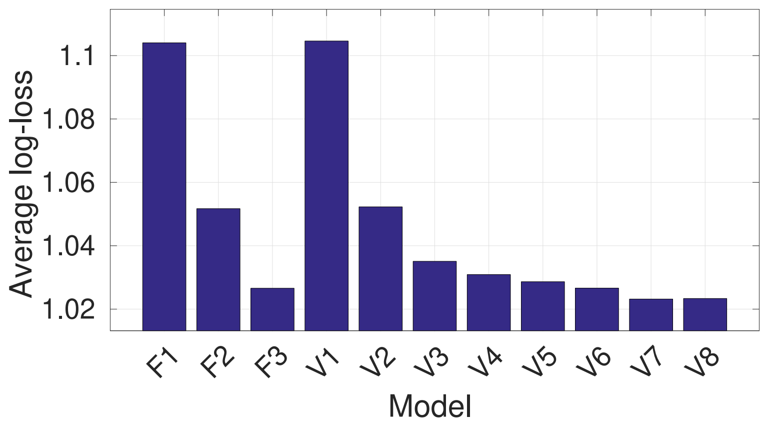
Formally, the symbolic regular expression that we use to capture the pattern of an increasing trend in the amount spent is the following:

$$\begin{aligned} R := \; & (\mathit{amountDiff} > 0) \cdot (\mathit{amountDiff} > 0) \cdot (\mathit{amountDiff} > 0) \cdot (\mathit{amountDiff} > 0) \cdot \\ & (\mathit{amountDiff} > 0) \cdot (\mathit{amountDiff} > 0) \cdot (\mathit{amountDiff} > 0) \end{aligned} \tag{7.7}$$

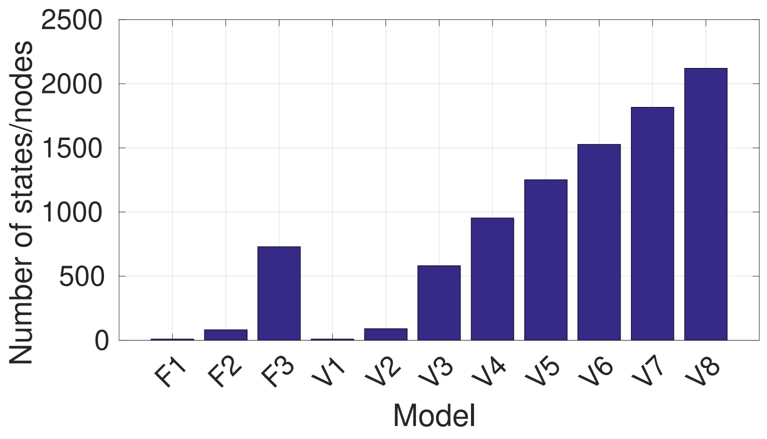
amountDiff is an extra attribute (besides the card ID, the amount spent, the transaction country and the other standard attributes) with which we enrich each event and is equal to the difference between the amount spent by the current transaction and that spent by the immediately previous transaction from the same card. The expression consists of seven terminal sub-expressions, in order to capture eight consecutive events. The first terminal sub-expression captures an increasing amount between the first two events in a fraudulent pattern.

If we attempted to perform forecasting based solely on Pattern (7.7), then the minterms that would be created would be based only on the predicate $\mathit{amountDiff} > 0$: namely, the predicate itself, along with its negation $\neg(\mathit{amountDiff} > 0)$. As expected, such an approach does not yield good results, as the language is not expressive enough to differentiate between fraudulent and legitimate transaction sequences. In order to address this lack of informative (for forecasting purposes) predicates, we have incorporated a mechanism in our system that allows us to incorporate extra predicates when building a probabilistic model, without affecting the semantics of the initial expression (exactly the same matches are detected). We do this by using any such extra predicates during the construction of the minterms. For example, if $\mathit{country} = \mathit{MA}$ is such an extra predicate that we would like included, then we would construct the following minterms for Pattern (7.7): a) $m_1 = (\mathit{amountDiff} > 0) \wedge (\mathit{country} = \mathit{MA})$; b) $m_2 = (\mathit{amountDiff} > 0) \wedge \neg(\mathit{country} = \mathit{MA})$; c) $m_3 = \neg(\mathit{amountDiff} > 0) \wedge (\mathit{country} = \mathit{MA})$; d) $m_4 = \neg(\mathit{amountDiff} > 0) \wedge \neg(\mathit{country} = \mathit{MA})$). We can then use these enhanced minterms as guards on the automaton transitions in a way that does not affect the semantics of the expression. For example, if an initial transition has the guard $\mathit{amountDiff} > 0$, then we can split it into two derived transitions, one for m_1 and one for m_2. The derived transitions would be triggered exactly when the initial one is triggered, the only difference being that the derived transitions also have information about the country. For our experiments and for Pattern (7.7), if we include the extra predicate $\mathit{amountDiff} > 100$, we expect the model to be able to differentiate between sequences involving genuine transactions (where the difference in the amount can by any value above 0) and fraudulent sequences (where the difference in the amount is always above 100 units).

We now present results for SDE forecasting. As already mentioned in Section 7.2.1, for this type of experiments we do not use the automaton created by Pattern (7.7). We instead use only its minterms which will constitute our "alphabet". In our case, there are four minterms: a) $\mathit{amountDiff} > 0 \wedge \mathit{amountDiff} > 100$; b) $\mathit{amountDiff} > 0 \wedge \neg(\mathit{amountDiff} > 100)$; c) $\neg(\mathit{amountDiff} > 0) \wedge \mathit{amountDiff} > 100$; d) $\neg(\mathit{amountDiff} > 0) \wedge \neg(\mathit{amountDiff} > 100)$. Thus, the goal is to predict, as the stream is consumed, which



(a) Average log-loss.

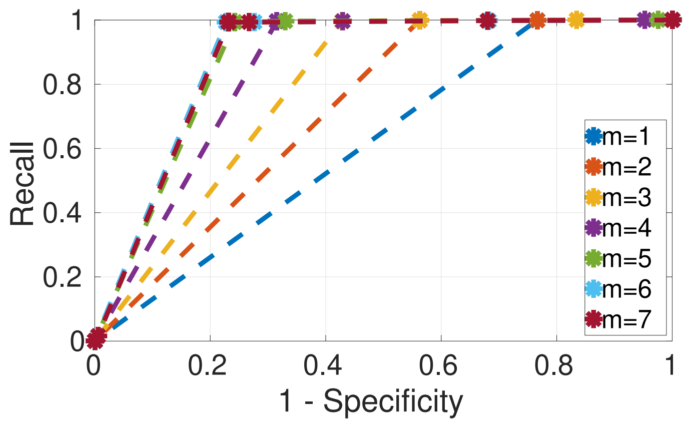


(b) Number of states/nodes.

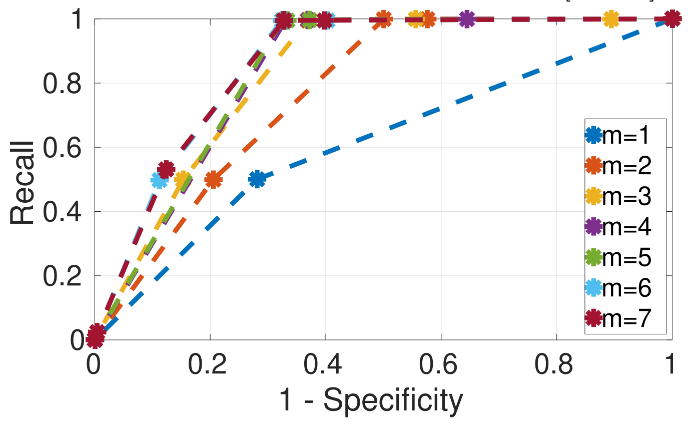
Figure 7.1: Results for SDE forecasting from the domain of credit card fraud management. Fx stands for a Full-order Markov Model of order x. Vx stands for a Variable-order Markov Model (a prediction suffix tree) of maximum order x.

one of these minterms will be satisfied. Notice that, for every possible event, exactly one minterm is satisfied (the third one, $\neg(\mathit{amountDiff} > 0) \wedge \mathit{amountDiff} > 100$, is actually unsatisfiable). We use 75% of the original dataset (which amounts to 750.000 transactions) for training and the rest (250.000) for testing. We do not employ cross-validation, as the dataset is synthetic and the statistical properties of its folds would not differ.

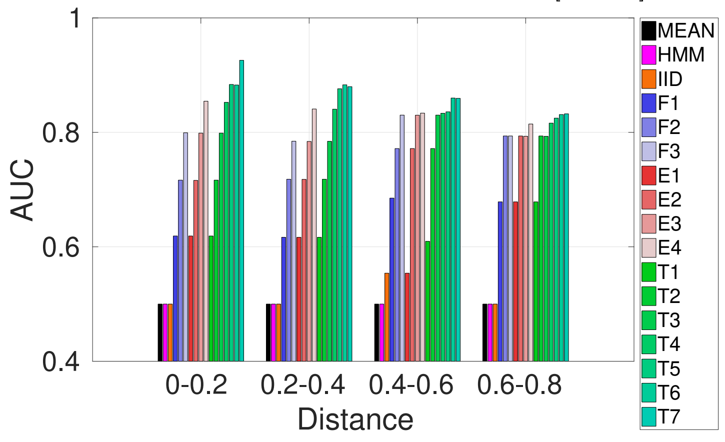
Figure 7.1a shows the average log-loss obtained for various models and orders m and Figure 7.1b shows the number of states for the full-order models or nodes for the variable-order models, which are prediction suffix trees. The best result is achieved with a variable-order Markov model of maximum order 7. The full-order Markov models are slightly better than their equivalent (same order) variable-order models. This is an expected behavior, since the variable-order models are essentially approximations of the full-order ones. We increase the order of the full-order models until $m = 3$, in which case the Markov chain has $\approx$ 750 states. We avoid increasing the order any further, because Markov chains with more than 1000 states become significantly difficult to manage in terms of memory usage (and in terms of the computational cost of estimating the waiting-time distributions for the experiments of CE forecasting that follow). Note that a Markov chain with 1000 states would require a transition matrix with 1000^2 entries. On the contrary, we can increase the maximum order of the variable-order model until we find the best one, i.e.,



(a) ROC curves for the variable-order model using the *PST* for various values of the maximum order m. *distance* $\in [0.2, 0.4]$.



(b) ROC curves for the variable-order model using the *PST* for various values of the maximum order. *distance* $\in [0.4, 0.6]$.



(c) AUC for ROC curves for all models.

Figure 7.2: Results for CE forecasting from the domain of credit card fraud management. Fx stands for a Full-order Markov Model of order x. Ex stands for a Variable-order Markov Model of maximum order x that uses a *PSA* and creates an embedding. Tx stands for a Variable-order Markov Model of maximum order x that is constructed directly from a *PST*. MEAN stands for the method of estimating the mean of "waiting-times". HMM stands for Hidden Markov Model. IID stands for the method assuming (in the worst case) that SDEs are i.i.d. Ex and Tx models are the ones proposed in this thesis.

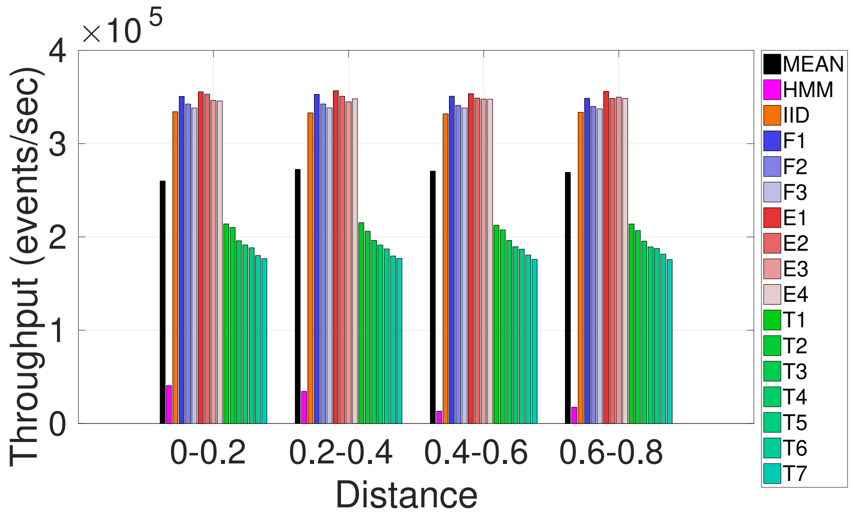
the order after which the average log-loss starts increasing again. The size of the prediction suffix tree can be allowed to increase to more than 1000 nodes, since we are not required to build a transition matrix.

We now move on to the classification experiments. Figure 7.2 shows the ROC curves of the variable-order model that directly uses a *PST*. We show results for two different "expected" distance ranges: $distance \in [0.2, 0.4]$ in Figure 7.2a and $distance \in [0.4, 0.6]$ in Figure 7.2b. The ideal operating point in the ROC is the top-left corner and thus, the closer to that point the curve is, the better. Thus, the first observation is that by increasing the maximum order we obtain better results. Notice, however, that in Figure 7.2b, where the distance is greater and the forecasting problem is harder, increasing the order from 6 to 7 yields only a marginal improvement.

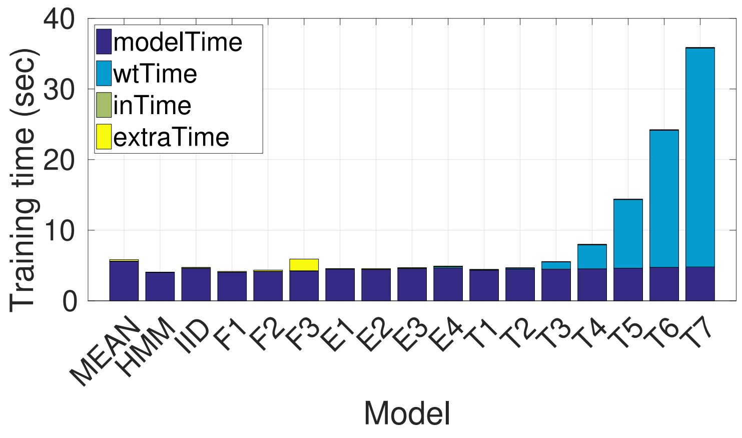
Figure 7.2c displays ROC results for different distances and all models, in terms of the Area Under the ROC Curve (AUC), which is a measure of the models' classification accuracy. The first observation is that the MEAN and HMM methods consistently underperform, compared to the Markov models. Focusing on the Markov models, as expected, the task becomes more challenging and the ROC scores decrease, as the distance increases. It is also evident that higher orders lead to better results. The advantage of increasing the order becomes less pronounced (or even non-existent) as the distance increases. The variable-order models that use an embedding are only able to go as far as $m = 4$, due to increasing memory requirements, whereas the tree-based versions can go up to $m = 7$ (and possibly even further, but we did not try to extend the order beyond this point). Although the embedding (*PSA*) can indeed help achieve better scores than full-order models by reaching higher orders, this is especially true for the tree-based models which bypass the embedding. We can thus conclude that full-order models are doing well up to the order that they we can achieve with them. *PSA* models can reach roughly the same levels, as they are also practically restricted. The performance of *PST* models is similar to that of the other models for the same order, but the fact that they can use higher orders allows them to finally obtain superior performance.

We show performance results in Figure 7.3, in terms of computation and memory efficiency. Figure 7.3a displays throughput results. We can observe the trade-off between the high forecasting accuracy of the tree-based high-order models and the performance penalty that these models incur. The models based on *PST* have a throughput figure that is almost half that of the full-order models and the embedding-based variable-order ones. In order to emit a forecast, the tree-based models need to traverse a tree after every new event arrives at the system, as described in Section 14. The automata-based full- and variable-order models, on the contrary, only need to evaluate the minterms on the outgoing transitions of their current state and simply jump to the next state. It would be possible to improve the throughput of the tree-based models, by using caching techniques, so that we can reuse some of the previously estimated forecasts, but we reserve such optimizations for future work. By far the worst throughput, however, is observed for the HMM models. The reason is that the waiting-time distributions and forecasts are always estimated online, as explained in Section 7.2.1.

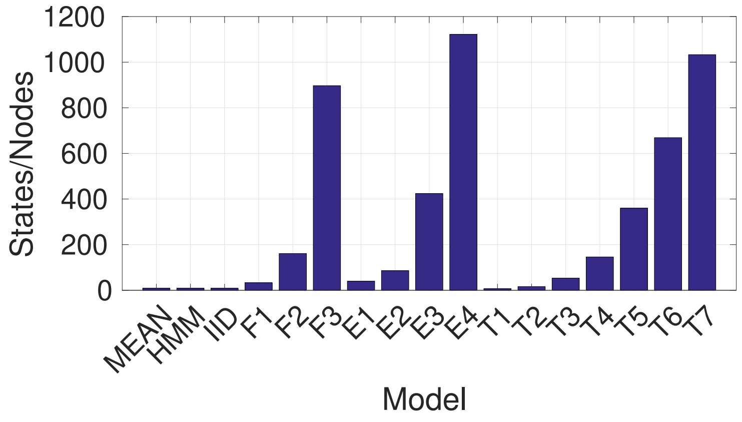
Figure 7.3b shows training times as a stacked, bar plot. For each model, the total training time is broken down into 4 different components, each corresponding to a different phase of the forecast building process. *modelTime* is the time required to actually construct the model from the training dataset. *wtTime* is the time required to estimate the waiting-time distributions, once the model has been constructed. *inTime* measures the time required to estimate the forecast of each waiting-time distribution. Finally, *extraTime* measures the



(a) Throughput.



(b) Training time.



(c) Number of states/nodes.

Figure 7.3: Results for throughput, training time and number of automaton states/tree nodes for classification CE forecasting from the domain of credit card fraud management. modelTime = time to construct the model. wtTime = time to estimate the waiting-time distributions for all states. inTime = time to estimate the forecast of all states from their waiting-time distributions. extraTime = time to determinize an automaton (+ disambiguation time for full-order models).

time required to determinize the automaton of our initial pattern. For the full-order Markov models, it also includes the time required to convert the deterministic automaton into its equivalent, disambiguated automaton. We observe that the tree-based models exhibit significantly higher times than the rest, for high orders. The other models have similar training times, almost always below 5 seconds. Thus, if we need high accuracy, we again have to pay a price in terms of training time. Even in the case of high-order tree-based models though, the training time is almost half a minute for a training dataset composed of 750,000 transactions, which allows us to be confident that training could be performed online.

Figure 7.3c shows the memory footprint of the models in terms of the size of their basic data structures. For automata-based methods, we show the number of states, whereas for the tree-based methods we show the number of nodes. We see that variable-order models, especially the tree-based ones, are significantly more compact than the full-order ones, for the same order. We also observe that the tree-based methods, for the same order, are much more compact (fewer nodes) than the ones based on the embedding (more states). This allows us to increase the order up to 7 with the tree-based approach, but only up to 4 with the embedding.

7.2.4 Maritime situational awareness

The second dataset that we used in our experiments is a real–world dataset coming from the field of maritime monitoring. It is composed of a set of trajectories from ships sailing at sea, emitting AIS (Automatic Identification System) messages that relay information about their position, heading, speed, etc., as described in the running example of Section 6.1. These trajectories can be analyzed, using the techniques of Complex Event Recognition, in order to detect interesting patterns in the behavior of vessels [113]. The dataset that we used is publicly available, contains AIS kinematic messages from vessels sailing in the Atlantic Ocean around the port of Brest, France, and spans a period from 1 October 2015 to 31 March 2016 [119]. We used a derivative dataset that contains clean and compressed trajectories, consisting only of critical points [114]. Critical points are the important points of a trajectory that indicate a significant change in the behavior of a vessel. Using critical points, one can reconstruct quite accurately the original trajectory [113]. We further processed the dataset by interpolating between the critical points in order to produce trajectories where two consecutive points have a temporal distance of exactly 60 seconds. The reason for this pre-processing step is that AIS messages typically arrive at unspecified time intervals. These intervals can exhibit a very wide variation, depending on many factors (e.g., human operators may turn on/off the AIS equipment), without any clear pattern that could be encoded by our probabilistic model. Consequently, our system performs this interpolation as a first step.

The pattern that we used in the experiments is a movement pattern in which a vessel approaches the main port of Brest. The goal is to forecast when a vessel will enter the port. This way, port traffic management may be optimized, in order to reduce the carbon emissions of vessels waiting to enter the port. The symbolic regular expression for this pattern is the following:

$$\begin{aligned} R := \; & (\neg InsidePort(Brest))^{*} \cdot (\neg InsidePort(Brest)) \cdot \\ & (\neg InsidePort(Brest)) \cdot (InsidePort(Brest)) \end{aligned} \tag{7.8}$$

The intention is to detect the entrance of a vessel in the port of Brest. The predicate $InsidePort(Brest)$ evaluates to TRUE whenever a vessel has a distance of less than 5 km

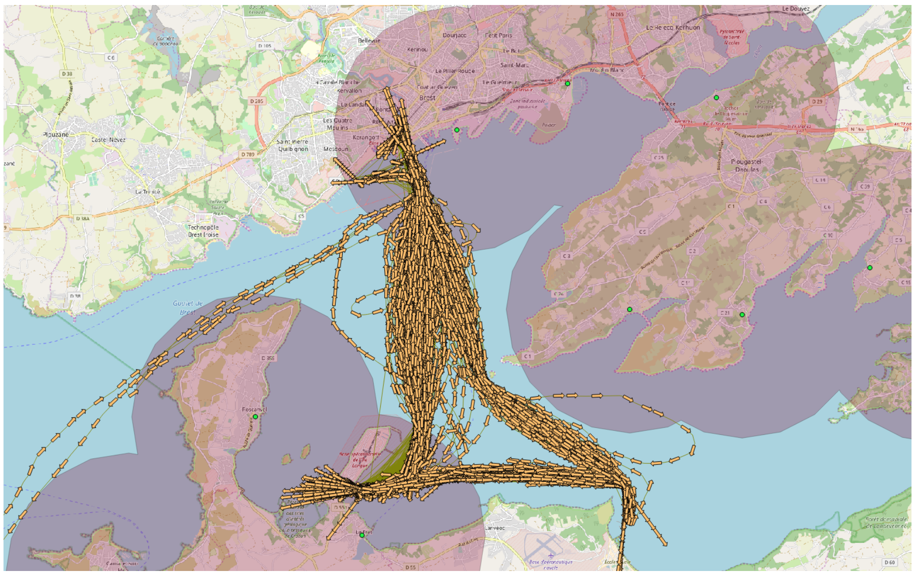


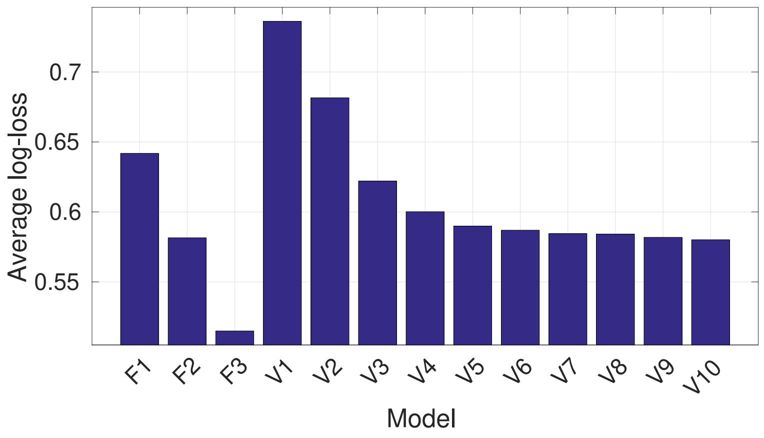
Figure 7.4: Trajectories of the vessel with the most matches for Pattern (7.8) around the port of Brest. Green points denote ports. Red, shaded circles show the areas covered by each port. They are centered around each port and have a radius of 5 km.

from the port of Brest (see Figure 7.4). In fact, the predicate is generic and takes as arguments the longitude and latitude of any point, but we show here a simplified version, using the port of Brest, for reasons of readability. The pattern defines the entrance to the port as a sequence of at least 3 consecutive events, only the last of which satisfies the $InsidePort(Brest)$ predicate. In order to detect an entrance, we must first ensure that the previous event(s) indicated that the vessel was outside the port. For this reason, we require that, before the last event, there must have occurred at least 2 events where the vessel was outside the port. We require 2 or more such events to have occurred (instead of just one), in order to avoid detecting "noisy" entrances.

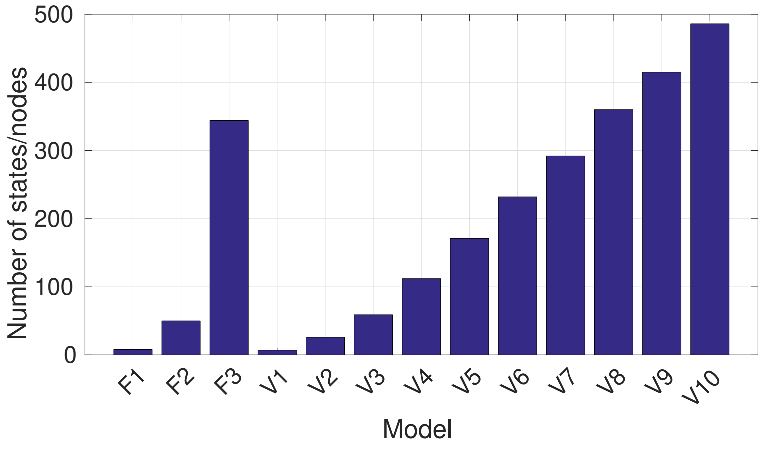
In addition to the $InsidePort(Brest)$ predicate, we included 5 extra ones providing information about the distance of a vessel from a port when it is outside the port. Each of these predicates evaluates to TRUE when a vessel lies within a specified range of distances from the port. The first returns TRUE when a vessel has a distance between 5 and 6 km from the port, the second when the distance is between 6 and 7 km and the other three extend similarly 1 km until 10 km. We investigated the sensitivity of our models to the presence of various extra predicates in the recognition pattern.

For all experimental results that follow, we always present average values over 4 folds of cross-validation. We start by analyzing the trajectories of a single vessel and then move to multiple, selected vessels. There are two issues that we tried to address by separating our experiments into single-vessel and multiple-vessel ones. First, we wanted to have enough data for training. For this reason, we only retained vessels for which we can detect a significant number of matches for Pattern (7.8). Second, our system can work in two modes: a) it can build a separate model for each monitored object and use this collection of models for personalized forecasting; b) it can build a global model out of all the monitored objects. We thus wanted to examine whether building a global model from multiple vessels could produce equally good results, as these obtained for a single vessel with sufficient training data.

We first used Pattern (7.8) to perform recognition on the whole dataset in order to



(a) Average log-loss.



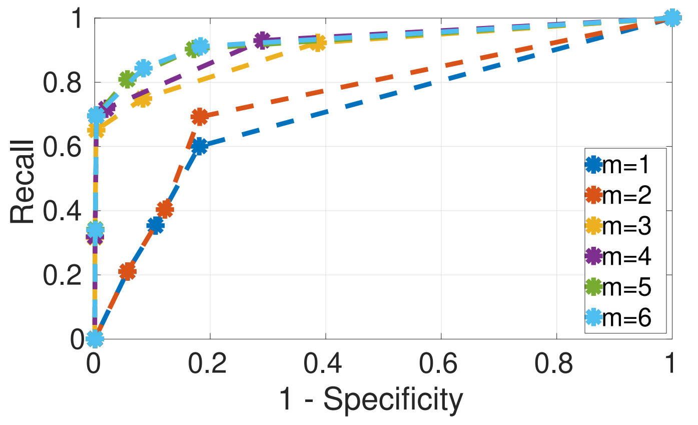
(b) Number of states/nodes.

Figure 7.5: Results for SDE forecasting from the domain of maritime monitoring. Fx stands for a Full-order Markov Model of order *x*. Vx stands for a Variable-order Markov Model (a prediction suffix tree) of maximum order *x*.

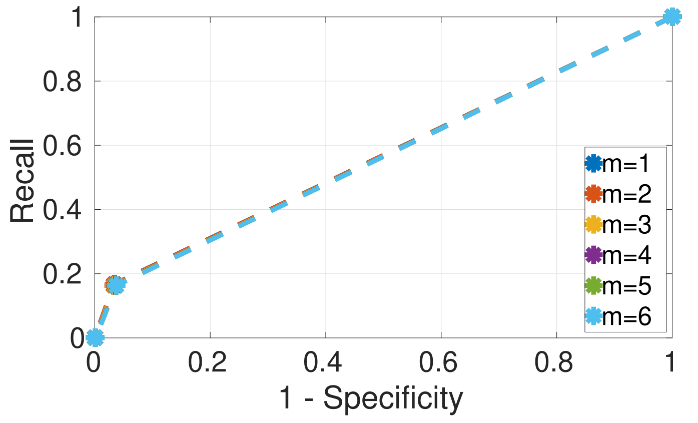
find the number of matches detected for each vessel. The vessel with the most matches was then isolated and we retained only the events emitted from this vessel. In total, we detected 368 matches for this vessel and the number of SDEs corresponding to it is $\approx$ 30.000. Figure 7.4 shows the isolated trajectories for this vessel, seemingly following a standard route between various ports around the Brest area.

Figure 7.5 shows results for SDE forecasting. The best average log-loss is achieved with a full-order Markov model, with $m = 3$, and is ≈ 0.51. For the best hyper-parameter values out of those that we tested for the variable-order model, with $m = 10$, we can achieve an average log-loss that is ≈ 0.57. Contrary to the case of credit card data, increasing the order of the variable-order model does not allow us to achieve a better log-loss score than the best one achieved with a full-oder model. However, as we will show, this does not imply that the same is true for CE forecasting.

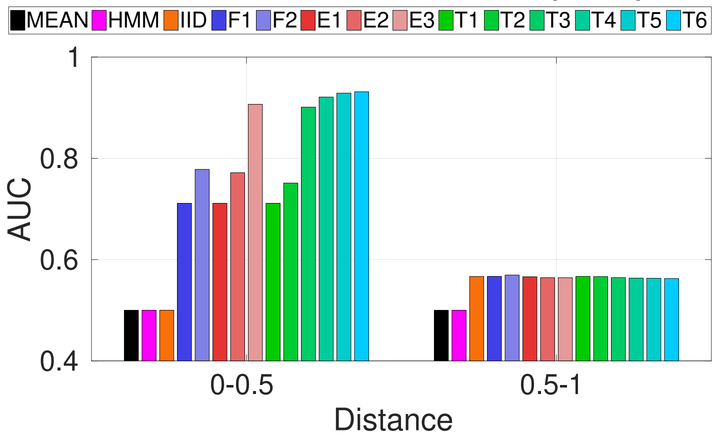
Using the vessel of Figure 7.4, we obtained the results shown in Figures 7.6 and 7.7. Since the original *DSFA* is smaller in this case (one start and one final state plus two intermediate states), we have fewer distance ranges (e.g., there no states in the range $[0.4, 0.6]$). Thus, we use only two distance ranges: $[0, 0.5]$ and $[0.5, 1]$. We observe the importance of being able to increase the order of our models for distances smaller than



(a) ROC curves for the variable-order model using the *PST* for various values of the maximum order. *distance* $\in [0.0, 0.5]$.

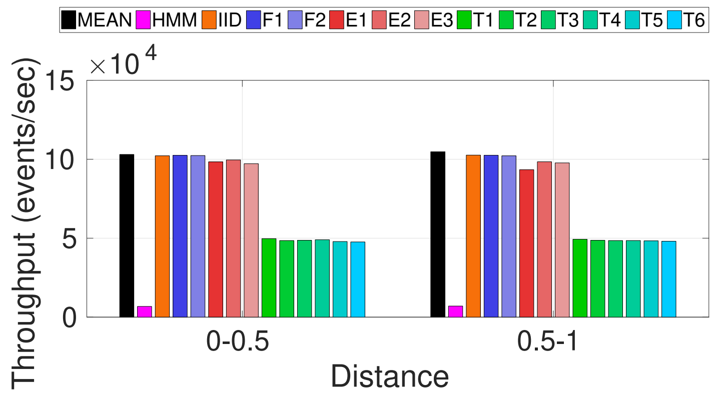


(b) ROC curves for the variable-order model using the *PST* for various values of the maximum order. *distance* $\in [0.5, 1.0]$.

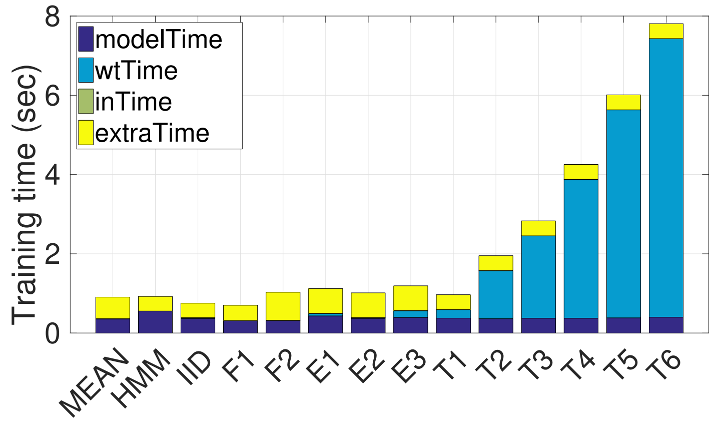


(c) AUC for ROC curves for all models.

Figure 7.6: Results for CE forecasting in the domain of maritime situational awareness. Fx stands for a Full-order Markov Model of order x. Vx stands for a Variable-order Markov Model of maximum order x using an embedding. Tx stands for a Variable-order Markov Model of maximum order x using a prediction suffix tree. MEAN stands for the method of estimating the mean of “waiting-times”. HMM stands for Hidden Markov Model. Ex and Tx models are the ones proposed in this thesis.



(a) Throughput.

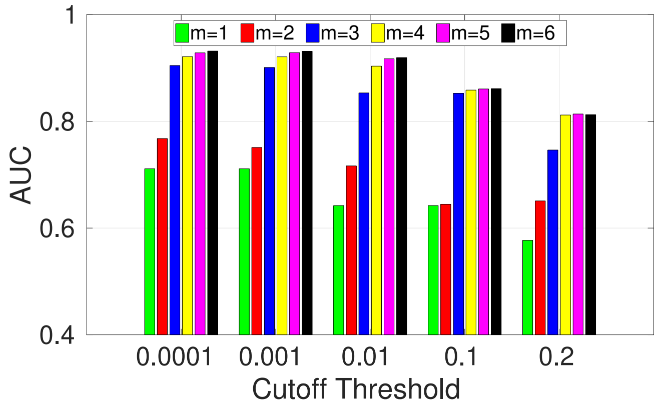


(b) Training time.

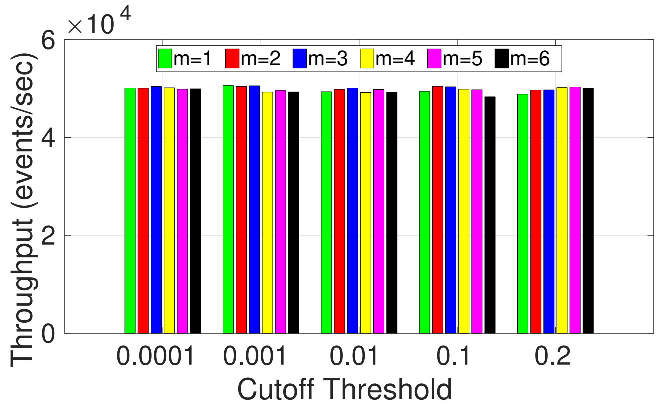
Figure 7.7: Throughput and training time results for classification CE forecasting for maritime situational awareness. modelTime = time to construct the model. wtTime = time to estimate the waiting-time distributions for all states. inTime = time to estimate the forecast interval of all states from their waiting-time distributions. extraTime = time to determinize an automaton (+ disambiguation time for full-order models).

50%. For distances greater than 50%, the area under curve is ≈ 0.5 for all models. This implies that they cannot effectively differentiate between positives and negatives. Their forecasts are either all positive, where we have $Recall = 100\%$ and $Specificity = 0\%$, or all negative, where we have $Recall = 0\%$ and $Specificity = 100\%$ (see Figure 7.6b). Notice that the full-order Markov models can now only go up to $m = 2$, since the existence of multiple extra predicates makes it prohibitive to increase the order any further. Achieving higher accuracy with higher-order models comes at a computational cost, as shown in Figure 7.7. The results are similar to those in the credit card experiments. The training time for variable-order models tends to increase as we increase the order, but is always less than 8 seconds. The effect on throughput is again significant for the tree-based variable-order models. Throughput figures are also lower here compared to the credit card fraud experiments, since the predicates that we need to evaluate for every new input event (like $InsidePort(Brest)$) involve more complex calculations (the $amountDiff > 0$ predicate is a simple comparison).

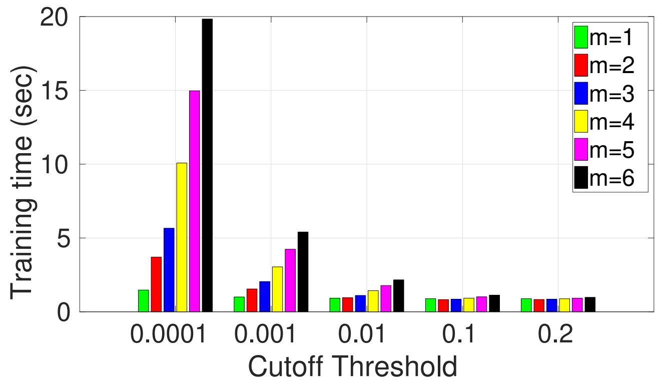
As a next step, we wanted to investigate the effect of the optimization technique mentioned at the end of Section 14 on the accuracy and performance of our system. The optimization prunes future paths whose probability is below a given cutoff threshold. We re-



(a) AUC-ROC.



(b) Throughput.



(c) Training time.

Figure 7.8: Effect of cutoff threshold on accuracy, throughput and training time.

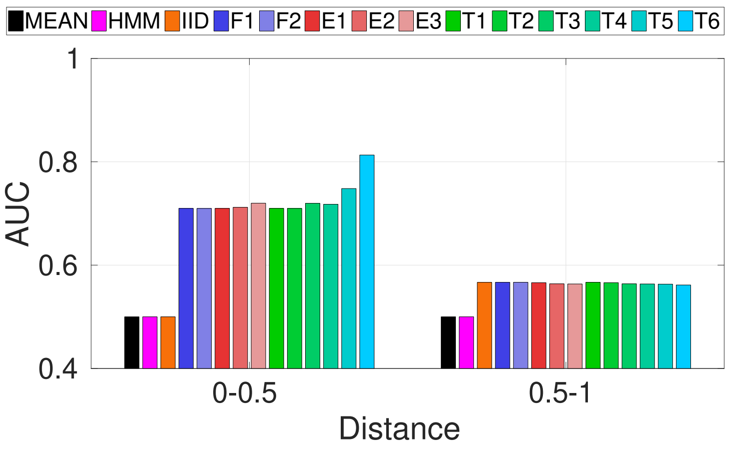
run the experiments described above for distances between 0% and 50% for various values of the cutoff threshold, starting from 0.0001 up to 0.2. Figure 7.8 shows the relevant results. We observe that the accuracy is affected only for high values of the cutoff threshold, above 0.1 (Figure 7.8a). We can also see that throughput remains essentially unaffected (Figure 7.8b). This result is expected, since the cutoff threshold is only used in the estimation of the waiting-time distributions. Throughput reflects the online performance of our system, after the waiting-time distributions have been estimated, and is thus not affected by the choice of the cutoff threshold. However, the training time is indeed significantly affected (Figure 7.8c). As expected, the result of increasing the value of the cutoff threshold is a reduction of the training time, as fewer paths are retained. Beyond a certain point though, further increases of the cutoff threshold affect the accuracy of the system. Therefore, the cutoff threshold should be below 0.01 so as not to compromise the accuracy of our forecasts.

We additionally investigated the sensitivity of our approach to the extra predicates that are used. Figure 7.9a shows results when the extra 5 predicates referring to the distance of a vessel from the port are modified so that each "ring" around the port has a width of 3 km, instead of 1 km. With these extra features, increasing the order does indeed make an important difference, but only when the order becomes high (5 and beyond), which is possible only by using the tree-based variable-order models. Moreover, the best score achieved is still lower than the best score achieved with "rings" of 1 km (Figure 7.6c). "Rings" of 1 km are thus more appropriate as predictive features. We also wanted to investigate whether other information about the vessel's movement could affect forecasting. In particular, we kept the 1 km "rings" and we also added a predicate to check whether the heading of a vessel points towards the port. More precisely, we used the vessel's speed and heading to project its location 1 hour ahead in the future and then checked whether this projected segment and the circle around the port intersect. The intuition for adding this feature is that the knowledge of whether a vessel is heading towards the port has predictive value. As shown in Figure 7.9b, this additional information led to higher scores even with low full-order orders (compare to Figure 7.6c). The heading feature is indeed important. On the other hand, the high-order models that did not use this feature seemed to be able to compensate for the missing information about the vessel's heading by going into higher orders. A plateau is thus reached, which cannot be "broken" with the heading information. Notice that here we can only go up to $m = 1$ for full-order models. The inclusion of the heading predicate leads to an increase of the number of states beyond 1200.

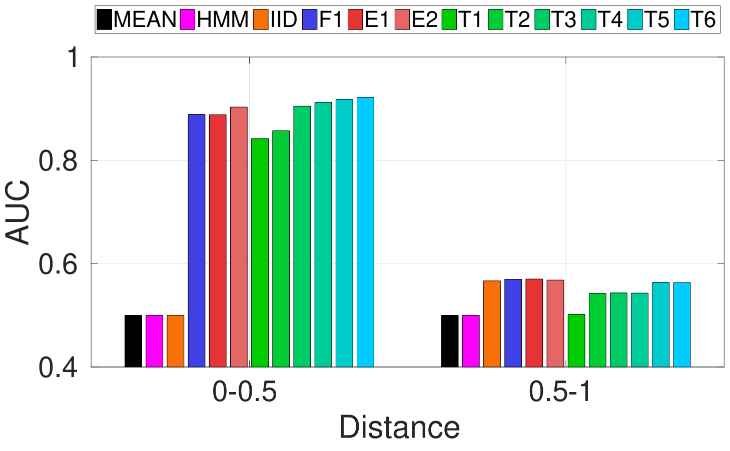
Finally, we also tested our method when more than one vessel need to be monitored. Instead of isolating the single vessel with the most matches, we isolated all vessels which had more than 100 matches. There are in total 9 such vessels in the dataset. The resulting dataset has ≈ 222.000 events. Out of the 9 retained vessels, we constructed a global probabilistic model and produced forecasts. An alternative option would be to build a single model for each vessel, but in this scenario we wanted to test the robustness of our aprroach when a global model is built from multiple entities. Figure 7.9c presents the corresponding results. Interestingly, the scores of the global model remain very close to the scores of the experiments for the single vessel with the most matches (Figure 7.6c). This is an indication of the ability of the global model to capture the peculiarities of individual vessels.

7.3 Summary

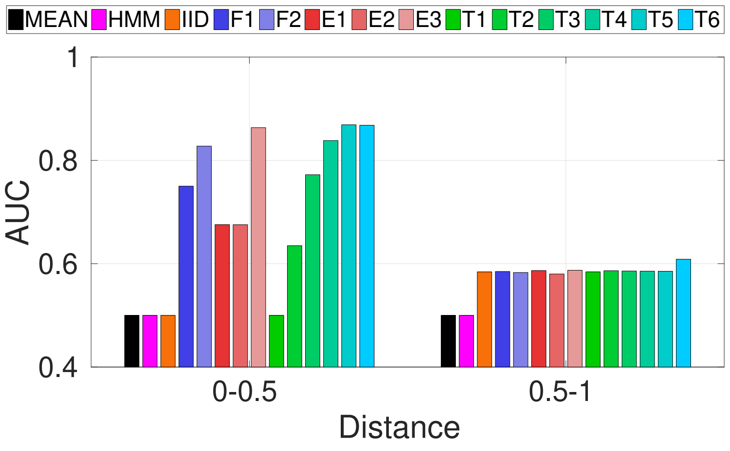
We have presented in the last two chapters a framework for Complex Event Forecasting (CEF), based on a variable-order Markov model. It allows us to delve deeper into the past and capture long-term dependencies, not feasible with full-order models. Our comprehensive evaluation on two application domains has illustrated the advantages of being able to use such high-order models. Namely, the use of higher-order modeling allows us to achieve higher accuracy than what is possible with full-order models or other state-of-the-art solutions. We have described two alternative ways in which variable-order models may be used, depending on the imposed requirements. One option is to use a highly efficient but less accurate model, when online performance is a top priority. We also provide an option that achieves high accuracy scores, but with a performance cost. Another important feature



(a) AUC for ROC curves. Extra features included: concentric rings around the port every 3 km. Single vessel.



(b) AUC for ROC curves. Extra features included: concentric rings around the port every 1 km and heading. Single vessel.



(c) AUC for ROC curves. Extra features included: concentric rings around the port every 1 km. Model constructed for the 9 vessels that have more than 100 matches.

Figure 7.9: Results for classification CE forecasting from the domain of maritime monitoring for various sets of extra features and for multiple vessels.

of our proposed framework is that it requires minimal intervention by the user. A given

Complex Event pattern is declaratively defined and subsequently automatically translated to an automaton and then to a Markov model, without requiring domain knowledge that should guide the modeling process.

Part Three

8. Symbolic regular expressions with memory

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 [42, 64, 93]. Typically, an event has the structure of a tuple of values which might be numerical or categorical. Since time is of critical importance for CER, 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 [56, 75, 93].

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 [3, 42, 46]. They resemble both register automata [79], through their ability to store events, and symbolic automata [44], 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 [67] 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 [67]. 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 [44]. 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 [88], where *Regular Expressions 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 [120, 121] 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 [9, 10]. We show here when and how *SRA* 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.

Our contributions may be summarized as follows:

Table 8.1: 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 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.
- 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 [10].

All proofs and complete algorithms may be found in Appendix B. Please, note that the results of this chapter 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 8.1** We now introduce an example which will be used throughout the chapter to provide intuition (borrowed from [67]). 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 8.1 shows an example of such a stream. We assume that events are temporally ordered and their order is implicitly provided through the index. ■

8.1 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 [52, 62], Cayuga [46], TESLA [40] and SASE [3, 143]. There also exist systems that do not employ automata as their computational model, e.g., there are logic-based systems [18] or systems that use trees [98], 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 [64], and for a general review of CER systems, see [42].

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 [67, 73]. Specifically, in [67], 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 [26, 79, 102]. For a review, see [125]. 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 [79]. At each transition, a register automaton may choose to store its current input (more precisely, the current input's data payload) to one of a finite set of registers. A transition is followed if the current input is equal to the contents of some register. With register automata, it is possible to recognize strings constructed from an infinite alphabet, through the use of (in)equality comparisons among the data carried by the current input and the data stored in the registers. However, register automata do not always have nice closure properties, e.g., they are not closed under determinization. For an extensive study of register automata, see [88, 89]. We build on the framework presented in [88, 89] 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 [104]. This initial idea has recently been expanded and more fully investigated in symbolic automata [44, 132, 133]. 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 [67, 68] 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 here. 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 [99], sequence prediction [24, 37, 121, 140], temporal mining [34, 83, 135, 144] and event sequence prediction and point-of-interest recommendations through neural networks [32, 87]. 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 [36, 53, 60]. In what follows, we briefly present the relatively few previous concrete attempts at CEF. The first such attempt at CEF was presented in [101], 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 [6, 8]. 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 [110], 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 ([2, 24]) 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 [86] 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.

8.2 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 [88], 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 8.2.1 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 8.2.2 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 8.2.3 we provide a precise definition for *SREM* and their semantics.

8.2.1 Formulas and models

In this section, we follow the notation and notions presented in [74]. The first notion that we need is that of a $\mathscr{V}$-structure. A $\mathscr{V}$-structure essentially describes a domain along with the operations that can be performed on the elements of this domain and their interpretation.

Definition 8.2.1 — $\mathscr{V}$-structure (74). A vocabulary V is a set of function, relation and constant symbols. A $\mathscr{V}$-structure is an underlying set $\mathscr{U}$, called a universe, and an interpretation of $\mathscr{V}$. An interpretation assigns an element of $\mathscr{U}$ to each constant in $\mathscr{V}$, a function from $\mathscr{U}^n$ to $\mathscr{U}$ to each n-ary function in $\mathscr{V}$ and a subset of $\mathscr{U}^n$ to each n-ary relation in $\mathscr{V}$. ◀

■ **Example 8.2** Using Example 8.1, we can define the following vocabulary

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

and the universe

$$\mathscr{U} = \{(T,1,22),(T,1,24),(T,2,32),(H,1,70),(H,1,68),(T,2,33)\}$$

We can also define an interpretation of V by assigning each c_i to an element of $\mathscr{U}$, 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 $\mathscr{U}$ 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 $\mathscr{U}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathscr{L}$ constructed from elements of $\mathscr{U}$ ($\mathscr{L} \subseteq \mathscr{U}^*$, where * denotes Kleene-star) is called a language over $\mathscr{U}$. 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 ($t_i \in \mathscr{U}$). 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 $\mathscr{V}$-structure. We begin with the definition of terms.

Definition 8.2.2 — Term (74). 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 8.2.3 — Formula (74). Let t_i be terms. A formula is defined as follows:
- If P is an n-ary relation, then $P(t_1, \cdots, t_n)$ is a formula (an atomic formula).
- If ϕ is a formula, $\neg\phi$ is also a formula.
- If ϕ_1 and ϕ_2 are formulas, $\phi_1 \wedge \phi_2$ is also a formula.
- If ϕ_1 and ϕ_2 are formulas, $\phi_1 \vee \phi_2$ is also a formula. ◀

Definition 8.2.4 — $\mathscr{V}$-formula (74). If $\mathscr{V}$ is a vocabulary, then a formula in which every function, relation and constant is in $\mathscr{V}$ is called a $\mathscr{V}$-formula. ◀

■ **Example 8.3** Continuing with our example, $R(c_1, c_4)$ is an atomic $\mathscr{V}$-formula. $R(c_1, c_4) \wedge \neg R(c_1, c_3)$ is also a (complex) $\mathscr{V}$-formula, where $\mathscr{V} = \{R, c_1, c_2, c_3, c_4, c_5, c_6\}$. ■

Notice that in typical definitions of terms and formulas (as found in [74]) 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 $\mathscr{U}$) 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 $\mathscr{V}$-structure.

Definition 8.2.5 — Model of $\mathscr{V}$-formulas (74). Let $\mathscr{M}$ be a $\mathscr{V}$-structure and ϕ a $\mathscr{V}$-formula. We define $\mathscr{M} \models \phi$ ($\mathscr{M}$ models ϕ) as follows:
- If ϕ is atomic, i.e. $\phi = P(t_1, \cdots, t_m)$, then $\mathscr{M} \models P(t_1, \cdots, t_m)$ iff the tuple $(a_1, \cdots, a_m)$ is in the subset of $\mathscr{U}^m$ assigned to P, where a_i are the elements of $\mathscr{U}$ assigned to the terms t_i.
- If $\phi := \neg\psi$, then $\mathscr{M} \models \phi$ iff $\mathscr{M} \not\models \psi$.
- If $\phi := \phi_1 \wedge \phi_2$, then $\mathscr{M} \models \phi$ iff $\mathscr{M} \models \phi_1$ and $\mathscr{M} \models \phi_2$.
- If $\phi := \phi_1 \vee \phi_2$, then $\mathscr{M} \models \phi$ iff $\mathscr{M} \models \phi_1$ or $\mathscr{M} \models \phi_2$. ◀

■ **Example 8.4** If $\mathscr{M}$ is the $\mathscr{V}$-structure of our example, then $\mathscr{M} \models R(c_1, c_4)$, since $c_1 \rightarrow (T, 1, 22)$, $c_4 \rightarrow (H, 1, 70)$ and $((T, 1, 22), (H, 1, 70)) \in R$. We can also see that $\mathscr{M} \models R(c_1, c_4) \wedge \neg R(c_1, c_3)$, since $c_3 \rightarrow (T, 2, 32)$ and $((T, 1, 22), (T, 2, 32)) \notin R$. ⊚ ■

8.2.2 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 8.2.6 — Condition. Let $\mathcal{M}$ be a $\mathcal{V}$-structure always equipped with the unary relation $\top$ for which it holds that $u \in \top, \forall u \in \mathcal{U}$, i.e., this relation holds for all elements of the universe $\mathcal{U}$. Let $R = \{r_1, \cdots, r_k\}$ be variables denoting the registers and $\sim$ a special variable denoting an automaton's head which reads new elements. The "contents" of the head always correspond to the most recent element. We call them register variables. A condition is essentially a $\mathcal{V}$-formula, as defined above (Definition 8.2.3), where, instead of terms, we use register variables. A condition is defined by the following grammar:

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

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

$$v'(r_j) = v[r_i \leftarrow u] = \begin{cases} u & \text{if } r_j = r_i \\ v(r_j) & \text{otherwise} \end{cases} \tag{8.1}$$

$v[W \leftarrow u]$, 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 ϕ if, for every register variable r_i that appears in ϕ, $v(r_i)$ 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 8.2.5. 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 8.2.8 — Semantics of conditions. Let $\mathcal{M}$ be a $\mathcal{V}$-structure, $u \in \mathcal{U}$ an element of the universe of $\mathcal{M}$ and $v \in F(r_1, \cdots, r_k)$ a valuation. We say that a condition ϕ is satisfied by (u, v), denoted by $(u, v) \models \phi$, iff one of the following holds:

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

8.2.3 Symbolic regular expressions with memory

We are now in a position to define Symbolic Regular Expressions with Memory *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 8.2.9 — Symbolic regular expression with memory (*SREM*). A symbolic regular expression with memory over a $\mathscr{V}$-structure $\mathscr{M}$ and a set of register variables $R = \{r_1, \cdots, r_k\}$ is inductively defined as follows:

1. ε and $\emptyset$ are *SREM*.
2. If ϕ is a condition (as in Definition 8.2.6), then ϕ is a *SREM*.
3. If ϕ is a condition, then $\phi \downarrow r_i$ is a *SREM*.
4. If e_1 and e_2 are *SREM*, then $e_1 + e_2$ is also a *SREM*.
5. If e_1 and e_2 are *SREM*, then $e_1 \cdot e_2$ is also a *SREM*.
6. If e is a *SREM*, then e^* is also a *SREM*. ◀

With *SREM* of the form $\phi \downarrow r_i$ (case 3 above), we denote cases where we need to store the current element read from the automaton's head to register r_i. Case 4 corresponds to the usual disjunction, whereas case 5 to concatenation. Finally, case 6 is the Kleene-star operator.

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

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

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

◀

In the first case, we have an ε *SREM*. It may reach another valuation only if it reads an ε string and this new valuation is the same as the initial one, i.e., the registers do not change. In the second case where we have a condition $\phi \neq \varepsilon$, 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. We accept the triggering element as part of the match only if the output indicates so. 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 [88]). 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 8.2.11 — 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 $(e,S,\sharp) \vdash v$, where $\sharp$ 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 $\mathscr{L}(e) = \{S \mid (e,S,\sharp) \vdash v\}$ for some valuation v. ◀

■ **Example 8.5** As an example, consider the following *SREM*

$$e_1 := (\mathit{TypeIsT}(\sim) \downarrow r_1) \cdot (\top)^* \cdot (\mathit{TypeIsH}(\sim) \wedge \mathit{EqualId}(\sim, r_1)) \tag{8.2}$$

where we assume that a) $\mathit{TypeIsT}(x) := x.type = T$, b) $\mathit{TypeIsH}(x) := x.type = H$ and c) $\mathit{EqualId}(x,y) := x.id = y.id$. If we feed the string/stream of Table 8.1 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 $v(r_1) = \sharp$ 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 $(\mathit{TypeIsH}(\sim) \wedge \mathit{EqualId}(\sim, r_1))$. 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 $S_{1..5}$. ■

9. Symbolic register automata

We now show how *SREM* can be translated to an appropriate automaton model and how this model may then be used to perform CER.

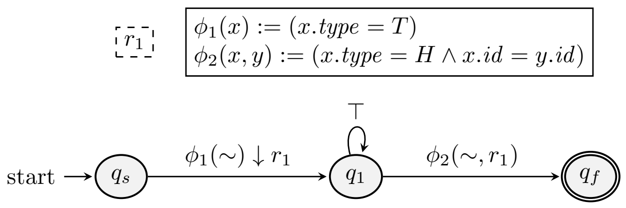
9.1 Symbolic register automata

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 ϵ-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 9.1.1 — Symbolic Register Automaton. A symbolic register automaton (*SRA*) with k registers over a $\mathscr{V}$-structure $\mathscr{M}$ is a tuple (Q, q_s, Q_f, R, Δ) where

- Q is a finite set of states,



r_1
$\phi_1(x) := (x.type = T)$
$\phi_2(x,y) := (x.type = H \wedge x.id = y.id)$
start
q_s
$\phi_1(\sim) \downarrow r_1$
q_1
$\top$
$\phi_2(\sim, r_1)$
q_f

Figure 9.1: *SRA* corresponding to Expression (8.2).

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

A transition $\delta \in \Delta$ is a tuple (q, ϕ, W, q'), also written as $q, \phi \downarrow W \rightarrow q'$, where
- $q, q' \in Q$,
- ϕ is a condition, as defined in Definition 8.2.6 or $\phi = \varepsilon$ 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 δ, we will also use the notation $\delta.source$ and $\delta.target$ to refer to its source and target states respectively.

■ **Example 9.1** As an example, consider the *SRA* of Figure 9.1. Each transition is represented as $\phi \downarrow W$, where ϕ 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 9.1). 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 ϕ_1. If ϕ_1 evaluates to true, it writes this last event to register r_1, displayed as a dashed square in Figure 9.1. On the other hand, the transition from q_1 to q_f uses both the current event and the event stored in r_1 (($\sim, r_1$)) and passes them to the binary formula ϕ_2. The condition $\top$ (in the self-loop of q_1) is a unary condition that always evaluates to true and allows us to skip any number of events. The *SRA* of Figure 9.1 captures *SREM* (8.2). ■

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

Definition 9.1.2 — 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:
- $\exists \delta : \delta.source = q,\ \delta.target = q',\ \delta.\phi = \varepsilon,\ j' = j,\ v' = v$, i.e., if this is an ε transition, we move to the target state without changing the index or the registers' contents.
- $\exists \delta : \delta.source = q,\ \delta.target = q',\ \delta.W = \emptyset,\ (t_j, v) \models \delta.\phi,\ j' = j + 1,\ v' = v,$

i.e., if the condition is satisfied according to the current event and the registers' contents and there are no write registers, we move to the target state, we increase the index by 1 and we leave the registers untouched.

- $\exists \delta : \delta.source = q,\ \delta.target = q',\ \delta.W \neq \emptyset,\ (t_j, v) \models \delta.\phi,\ j' = j+1,\ v' = v[W \leftarrow t_j]$, i.e., if the condition is satisfied according to the current event and the registers' contents and there are write registers, we move to the target state, we increase the index by 1 and we replace the contents of all write registers (all $r_i \in W$) with the current element from the string. ◀

We denote a succession by $[j,q,v] \rightarrow [j',q',v']$, or $[j,q,v] \overset{\delta}{\rightarrow} [j',q',v']$ if we need to refer to the transition as well. For the initial configuration, before any elements have been consumed, we assume that $j = 1$, $q = q_s$ and $v(r_i) = \sharp$, $\forall r_i \in R$. In order to move to a successor configuration, we need a transition whose condition evaluates to true, when applied to $\sim$, if it is unary, or to $\sim$ and the contents of its register selection, if it is n-ary. If this is the case, we move one position ahead in the stream and update the contents of this transition's write registers, if any, with the event that was read. If the transition is an ϵ-transition, we do not move the stream pointer 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 9.1.3 — Run of *SRA* over string/stream. A run ρ of a *SRA* A over a stream $S = t_1, \cdots, t_n$ is a sequence of successor configurations $[1, q_1, v_1] \overset{\delta_1}{\rightarrow} [2, q_2, v_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_n}{\rightarrow} [n+1, q_{n+1}, v_{n+1}]$. A run is called accepting iff $q_{n+1} \in A.Q_f$. ◀

■ **Example 9.2** A run of the *SRA* of Figure 9.1, while consuming the first four events from the stream of Table 8.1, is the following:

$$[1, q_s, \sharp] \overset{\delta_{s,1}}{\rightarrow} [2, q_1, (T,1,22)] \overset{\delta_{1,1}}{\rightarrow} [3, q_1, (T,1,22)] \overset{\delta_{1,1}}{\rightarrow} [4, q_1, (T,1,22)] \overset{\delta_{1,f}}{\rightarrow} [5, q_f, (T,1,22)] \tag{9.1}$$

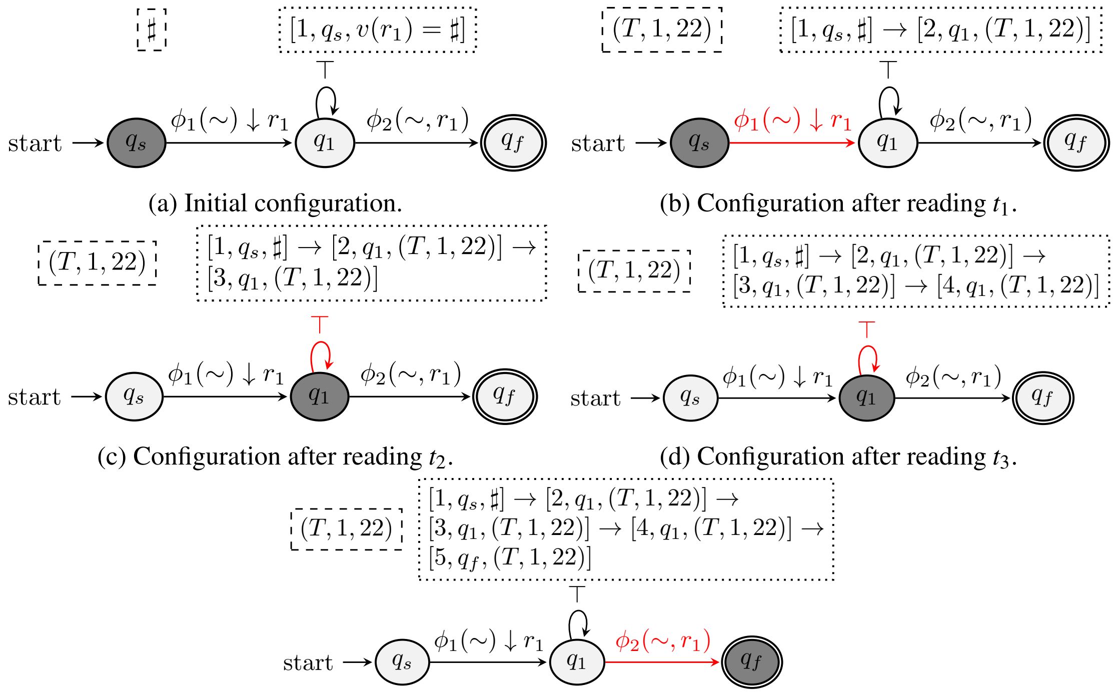
Transition subscripts in this example refer to states of the *SRA*, e.g., $\delta_{s,s}$ is the transition from the start state to itself, $\delta_{s,1}$ is the transition from the start state to q_1, etc. See also Figure 9.2. Run (9.1) is not the only run, since the *SRA* could have followed other transitions 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 9.1.4 — Language recognized by *SRA*. We say that a *SRA* A accepts a string S iff there exists an accepting run $\rho = [1, q_1, v_1] \overset{\delta_1}{\rightarrow} [2, q_2, v_2] \overset{\delta_2}{\rightarrow} \cdots \overset{\delta_n}{\rightarrow} [n+1, q_{n+1}, v_{n+1}]$ of A over S, where $q_1 = A.q_s$ and $v_1 = \sharp$. The set of all strings accepted by A is called the language recognized by A and is denoted by $\mathscr{L}(A)$. ◀

9.2 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


♯
[1, q_s, $v(r_1)$ = ♯]
start
q_s
$\phi_1(\sim) \downarrow r_1$
$\top$
q_1
$\phi_2(\sim, r_1)$
q_f
(T, 1, 22)
[1, q_s, ♯] → [2, q_1, (T, 1, 22)]
[1, q_s, ♯] → [2, q_1, (T, 1, 22)] → [3, q_1, (T, 1, 22)]
[1, q_s, ♯] → [2, q_1, (T, 1, 22)] → [3, q_1, (T, 1, 22)] → [4, q_1, (T, 1, 22)]
[1, q_s, ♯] → [2, q_1, (T, 1, 22)] → [3, q_1, (T, 1, 22)] → [4, q_1, (T, 1, 22)] → [5, q_f, (T, 1, 22)]

(a) Initial configuration.

(b) Configuration after reading t_1.

(c) Configuration after reading t_2.

(d) Configuration after reading t_3.

(e) Configuration after reading t_4.

Figure 9.2: A run of the *SRA* of Figure 9.1, while consuming the first four events from the stream of Table 8.1. Triggered transitions are shown in red and the current state of the *SRA* in dark gray. The dashed box represents a register. The contents of the register at each configuration are shown inside the dashed box. Inside the dotted boxes, the run is shown.

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. 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.

9.2.1 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., $\mathscr{L}(e) = \mathscr{L}(A)$. See Definitions 8.2.11 and 9.1.4.

Theorem 9.2.1 For every *SREM* e there exists an equivalent *SRA* A, i.e., a *SRA* such that $\mathscr{L}(e) = \mathscr{L}(A)$.

Proof. The complete *SRA* construction process and proof may be found in Appendix B.1. ■

■ **Example 9.3** Here, we present an example, to give the intuition. Let

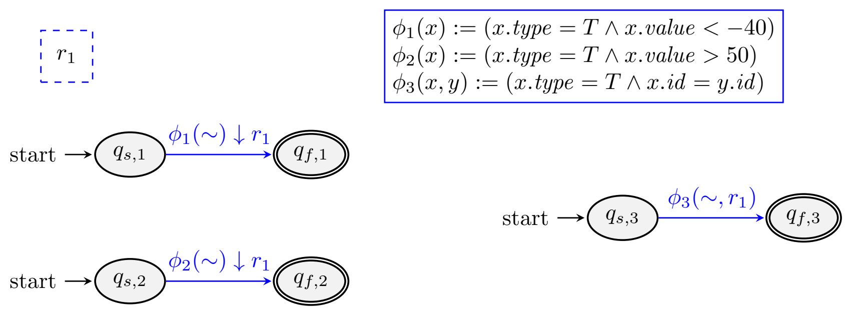
$$\begin{aligned} e_2 :=&((\phi_1(\sim) \downarrow r_1) + (\phi_2(\sim) \downarrow r_1))\cdot \\ &(\phi_3(\sim, r_1)) \end{aligned} \tag{9.2}$$

be a *SREM*, where

$$\begin{aligned} \phi_1(x) :=&(x.type = T \wedge x.value < -40) \\ \phi_2(x) :=&(x.type = T \wedge x.value > 50) \\ \phi_3(x,y) :=&(x.type = T \wedge x.id = y.id) \end{aligned}$$

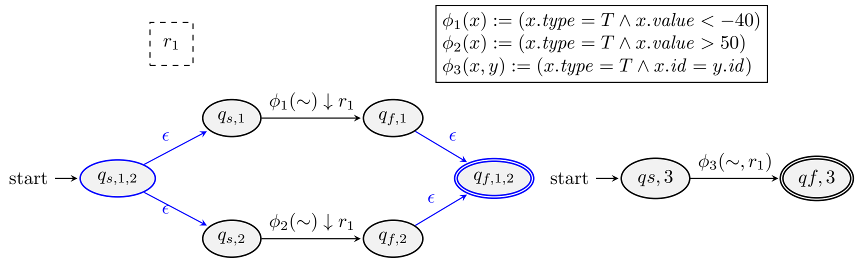
With this expression, we want to monitor sensors for possible failures. We want to detect cases where a sensor records temperatures outside some range of values (first line of *SREM* (9.2)) and continues to transmit measurements (second line), so that we are alerted to the fact that new measurements might not be trustworthy. The last condition is a binary formula, applied to both $\sim$ and r_1. Figure 9.3 shows the process for constructing the *SRA* which is equivalent to *SREM* (9.2).

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 *SREM* e, i.e. any registers in the register selection of any transitions and any write registers. We first create those registers and then



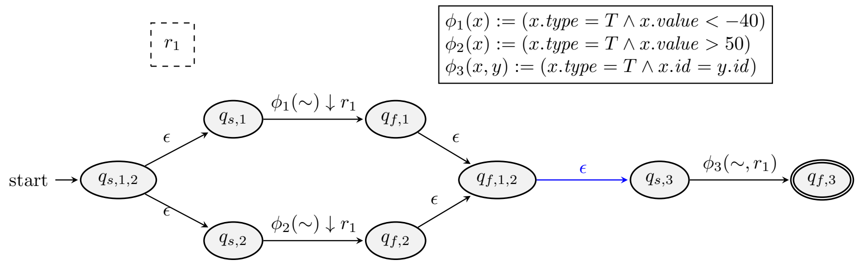
r_1
$\phi_1(x) := (x.type = T \land x.value < -40)$
$\phi_2(x) := (x.type = T \land x.value > 50)$
$\phi_3(x, y) := (x.type = T \land x.id = y.id)$
start
$q_{s,1}$
$\phi_1(\sim) \downarrow r_1$
$q_{f,1}$
start
$q_{s,3}$
$\phi_3(\sim, r_1)$
$q_{f,3}$
start
$q_{s,2}$
$\phi_2(\sim) \downarrow r_1$
$q_{f,2}$

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



r_1
$\phi_1(x) := (x.type = T \land x.value < -40)$
$\phi_2(x) := (x.type = T \land x.value > 50)$
$\phi_3(x, y) := (x.type = T \land x.id = y.id)$
start
$q_{s,1,2}$
ϵ
$q_{s,1}$
$\phi_1(\sim) \downarrow r_1$
$q_{f,1}$
$q_{s,2}$
$\phi_2(\sim) \downarrow r_1$
$q_{f,2}$
$q_{f,1,2}$
start
$qs, 3$
$\phi_3(\sim, r_1)$
$qf, 3$

(b) Connecting *SRA* via disjunction.



r_1
$\phi_1(x) := (x.type = T \land x.value < -40)$
$\phi_2(x) := (x.type = T \land x.value > 50)$
$\phi_3(x, y) := (x.type = T \land x.id = y.id)$
start
$q_{s,1,2}$
ϵ
$q_{s,1}$
$\phi_1(\sim) \downarrow r_1$
$q_{f,1}$
$q_{s,2}$
$\phi_2(\sim) \downarrow r_1$
$q_{f,2}$
$q_{f,1,2}$
$q_{s,3}$
$\phi_3(\sim, r_1)$
$q_{f,3}$

(c) Connecting *SRA* via concatenation.

Figure 9.3: Constructing *SRA* from *SREM* (9.2). New elements added at every step are shown in blue.

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 9.3, 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 9.3a. To the first two transitions, we add the relevant *unary* conditions, e.g., we add $\phi_1(x){:=}(x.type{=}T \land x.value{<}-40)$ to $q_{s,1}{\rightarrow}q_{f,1}$. To the third transition, we add the relevant *binary* condition $\phi_3(x,y) := (x.type = T \land x.id = y.id)$. The $+$ operator is handled by joining the *SRA* of the disjuncts through new states and ε-transitions. See Figure 9.3b. The concatenation operator is handled by connecting the *SRA* of its sub-expressions through an ε-transition, without adding any new states. See Figure 9.3c. Iteration, not applicable in this example, is handled by joining the final state of the original automaton to its start state through an ε-transition. ■

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 ε elimination, stating that we can always eliminate all ε transitions from a *SRA* to get an equivalent *SRA* with no ε transitions.

Lemma 9.2.2 For every *SRA* A_{ε} with ε transitions there exists an equivalent *SRA* $A_{\notin}$ without ε transitions, i.e., a *SRA* such that $\mathscr{L}(A_{\varepsilon}) = \mathscr{L}(A_{\notin})$.

Proof. See Appendix B.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 δ in Definition 9.1.1, $\delta.W$, might not be a singleton. On the other hand, according to Definition 8.2.9, 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 9.2.1 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 9.2.3 For every multi-register *SRA* A_{mr} there exists an equivalent single-register *SRA* A_{sr}, i.e., a single-register *SRA* such that $\mathscr{L}(A_{mr}) = \mathscr{L}(A_{sr})$.

Proof. See Appendix B.3. ■

We are now in a position to prove that every *SRA* can be converted to a *SREM*.

Theorem 9.2.4 For every *SRA* A there exists an equivalent *SREM* e, i.e., a *SREM* such that $\mathscr{L}(A) = \mathscr{L}(e)$.

Proof. See Appendix B.4. ■

9.2.2 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 9.2.2 — Deterministic *SRA* (*dSRA*). A *SRA* A with k registers $\{r_1, \cdots, r_k\}$ over a $\mathscr{V}$-structure $\mathscr{M}$ 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 $u \in \mathscr{M}.\mathscr{U}$ and $v \in F(r_1, \cdots, r_k)$, $(u,v) \models \phi_1$ and $(u,v) \models \phi_2$ cannot both hold, i.e.,

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

◀

In other words, from all the outgoing transitions from a given state q at most one of them can be triggered on any element u and valuation/register contents v. By definition, for a deterministic *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 9.2.3 — 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 $\mathscr{L}(\mathscr{A}) = \mathscr{L}(A_1) \cup \mathscr{L}(A_2)$, 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 $\mathscr{L}(\mathscr{A}) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$, 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 $\mathscr{L}(A) = \mathscr{L}(A_1) \cdot \mathscr{L}(A_2)$, 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 $\mathscr{L}(A_*) = (\mathscr{L}(A))^*$, where $L^* = \bigcup\limits_{i \geq 0} L^i$, 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 $S \in \mathscr{L}(A) \Leftrightarrow S \notin \mathscr{L}(A_c)$.
- determinization if, for every *SRA* A, there exists a *dSRA* A_D such that $\mathscr{L}(A) = \mathscr{L}(A_d)$.

◀

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

Theorem 9.2.5 *SRA* and *SREM* are closed under union, intersection, concatenation and Kleene-star.

Proof. See Appendix B.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 δ to $q' = (q'_1, q'_2) \in Q$ if there exists a transition δ_1 from q_1 to q'_1 in A_1 and a transition δ_2 from q_2 to q'_2 in A_2. The write registers of δ 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 9.2.6 *SRA* and *SREM* are not closed under complement.

Proof. See Appendix B.6. ■

It is also not always possible to determinize them:

Theorem 9.2.7 *SRA* are not closed under determinization.

Proof. See Appendix B.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 [79], *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.

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

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*.

9.2.3 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 9.2.4 — Windowed *SREM*. Let e be a *SREM* over a $\mathcal{V}$-structure $\mathcal{M}$ and a set of register variables $R = \{r_1, \cdots, r_k\}$, S a string constructed from elements of the universe of $\mathcal{M}$ and $v, v' \in F(r_1, \cdots, r_k)$. A windowed *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 9.2.8 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 B.9. ■

■ **Example 9.4** Here, we provide only the general outline of the algorithm and an example. Consider, e.g., the following *SREM*:

$$e_3 := ((\top)^* \cdot TypeIsT(\sim) \downarrow r_1) \cdot (\top)^* \cdot (TypeIsH(\sim) \wedge EqualId(\sim, r_1)))^{[1..w]} \quad (9.3)$$

Algorithm 4: Constructing unrolled *SRA* for windowed *SREM* (simplified).

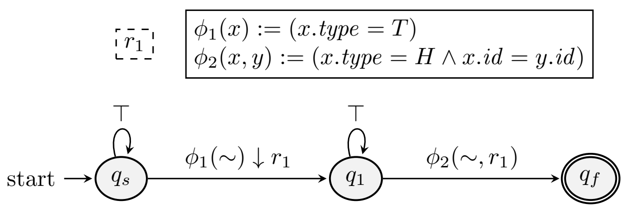
```
Input: Windowed SREM $e' := e^{[1..w]}$
Output: Deterministic SRA $A_{e'}$ equivalent to $e'$
1 $A_{e,\varepsilon} \leftarrow ConstructSRA(e)$;
2 $A_e \leftarrow EliminateEpsilon(A_{e,\varepsilon})$;
3 enumerate all walks of $A_e$ of length up to $w$; // Now unroll $A_e$.
4  join walks through disjunction;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

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 $TypeIsT(\sim) \downarrow r_1$). 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 ($TypeIsH(\sim) \wedge EqualId(\sim, r_1)$), 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 9.4b shows the steps taken for constructing the equivalent unrolled *SRA* for this expression. A simplified version of the unrolling algorithm is shown in Algorithm 4.

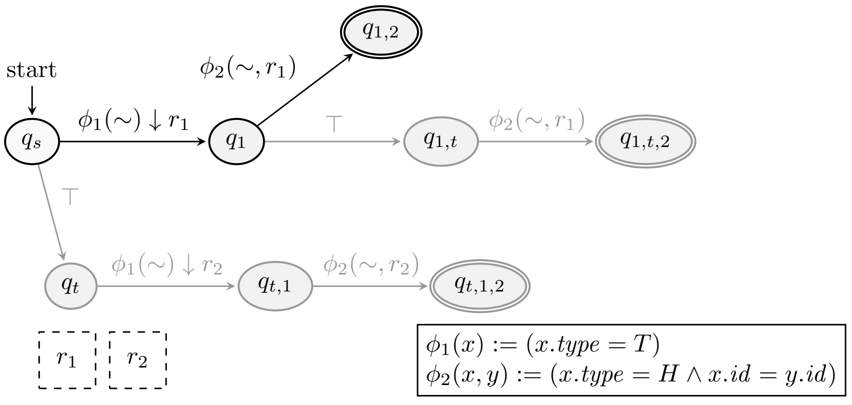
The construction algorithm first produces a *SRA* as usual, without taking the window operator into account (see line 1 of Algorithm 4). For our example, the result would be the *SRA* of Figure 9.4a (please, note that the automaton of Figure 9.4a is slightly different than that of Figure 9.1 due to the presence of the first $(\top)^*$ sub-expression). Then the algorithm eliminates any ε-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 4 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 4 show this process in a simplified manner. The *uSRA* for our example is shown in Figure 9.4b 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 9.4a, also named q_s in Figure 9.4b. 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 ϕ_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 9.4b. 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 9.2.8.

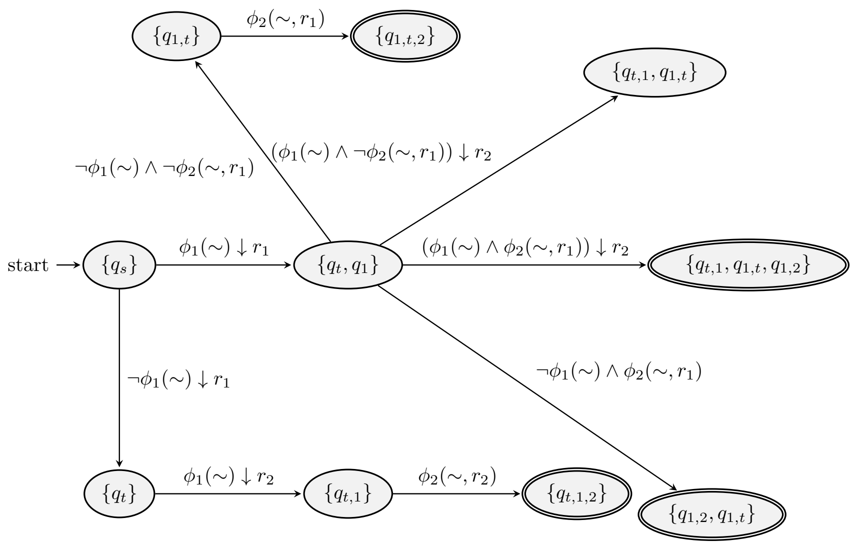
Having a *uSRA* makes it easy to subsequently construct a *dSRA*:


r_1
$\phi_1(x) := (x.type = T)$
$\phi_2(x,y) := (x.type = H \wedge x.id = y.id)$
start
$\top$
q_s
$\phi_1(\sim) \downarrow r_1$
q_1
$\phi_2(\sim, r_1)$
q_f

(a) *SRA* for *SREM* (9.3) before unrolling.


start
q_s
$\phi_1(\sim) \downarrow r_1$
q_1
$\phi_2(\sim, r_1)$
$q_{1,2}$
$\top$
$q_{1,t}$
$q_{1,t,2}$
q_t
$\phi_1(\sim) \downarrow r_2$
$q_{t,1}$
$\phi_2(\sim, r_2)$
$q_{t,1,2}$
r_1
r_2
$\phi_1(x) := (x.type = T)$
$\phi_2(x,y) := (x.type = H \wedge x.id = y.id)$

(b) *SRA* for *SREM* (9.3) after unrolling cycles, for $w = 3$ (whole *SRA*, black and light gray states) and $w = 2$ (top 3 states in black).


$\{q_{1,t}\}$
$\phi_2(\sim, r_1)$
$\{q_{1,t,2}\}$
$\{q_{t,1}, q_{1,t}\}$
$\neg\phi_1(\sim) \wedge \neg\phi_2(\sim, r_1)$
$(\phi_1(\sim) \wedge \neg\phi_2(\sim, r_1)) \downarrow r_2$
start
$\{q_s\}$
$\phi_1(\sim) \downarrow r_1$
$\{q_t, q_1\}$
$(\phi_1(\sim) \wedge \phi_2(\sim, r_1)) \downarrow r_2$
$\{q_{t,1}, q_{1,t}, q_{1,2}\}$
$\neg\phi_1(\sim) \downarrow r_1$
$\neg\phi_1(\sim) \wedge \phi_2(\sim, r_1)$
$\{q_t\}$
$\phi_1(\sim) \downarrow r_2$
$\{q_{t,1}\}$
$\phi_2(\sim, r_2)$
$\{q_{t,1,2}\}$
$\{q_{1,2}, q_{1,t}\}$

(c) *dSRA* (only part of it), for $w = 3$.

Figure 9.4: Constructing *dSRA* for *SREM* (9.3).

Theorem 9.2.9 For every windowed *SREM* there exists an equivalent deterministic *SRA*.

Proof. The proof for determinization is presented in Appendix B.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 [44, 133]. 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 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 9.5** As an example, Figure 9.4c shows the result of converting the *uSRA* of Figure 9.4b to a *dSRA*. We have simplified somewhat the conditions of each transition due to the presence of the $\top$ predicates in some of them. For example, the minterm $\phi_1 \wedge \neg \top$ for the start state is unsatisfiable and can be ignored while $\phi_1 \wedge \top$ may be simplified to ϕ_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 $\phi_2(\sim, r_1) \wedge \neg \phi_2(\sim, r_2)$, 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 9.2.10 Windowed *SRA* are closed under complement.

Proof. See Appendix B.11. ■

9.3 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 [10], 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 Σ is the automaton's (classical) alphabet. For example, if we apply $R := \{a, b, c\}^* \cdot (a \cdot b)$ on the stream a, b, c, a, b, c, the corresponding automaton would indeed reach its final state after reading the second and the fifth symbols. In our case, events come in the form of tuples with both numerical and categorical values. Using database systems terminology we can speak of tuples from relations of a database schema [67]. These tuples constitute the universe $\mathscr{U}$ 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 9.3.1 — 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 9.3.1 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 B.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.

R The above definition defines when a complex event is said to occur. However, it is customary in the CER literature to define complex events as sets of input events, called the matches of a given pattern.

■ **Example 9.6** For example, assume we have the following *SREM*:

$$e_4 := (\phi_1(\sim) \downarrow r_1) \cdot (\phi_2(\sim, r_1))$$

where

$$\phi_1(x) := (x.type = T)$$
$$\phi_2(x,y) := (x.type = T \wedge x.id = y.id \wedge x.value > y.value)$$

If we use this *SREM* on the stream of Table 8.1, then, after the second event, we would have the match $\{1,2\}$, where we indicate events by their index. After the third event, however, we would have two matches: $\{1,3\}$ and $\{2,3\}$. ■

In other words, we may need to know not only when a complex event occurs, but also which simple events participated in its detection. For this functionality, we would need to use transducers which have the ability to produce output on their transitions and mark some of the input events as participating in a match and others as irrelevant (see [67] for such an approach, using symbolic transducers, but without memory). In this thesis, we avoid discussing this issue and assume that we are only interested in when a complex event occurs. Ignoring the output of transitions is a reasonable restriction for forecasting, since we are only interested about when a pattern is detected and not about which specific input events constitute a match. Notice, however, that using transducers does not significantly affect our results [7]. The problem in this case is that windowed transducers with memory are only determinizable if we ignore the output of their transitions. This result is expected, since deterministic transducers have only a single run and cannot differentiate between different matches.

■ **Example 9.7** In our example above, we would need two different runs, one that would report $\{1,3\}$ and another one for $\{2,3\}$. ■

If we do need to perform both CER with match reporting and CEF, the solution would then be to use a non-deterministic transducer for CER and, at the same time, its deterministic version for CEF.

9.4 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 $EqualId(\sim, r_1)$ condition in *SREM* (9.3). 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 [88]). 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).

R We intend to investigate in the future the possibility of providing even more precise complexity results for *SREM* and *SRA*, both from the point of view of formal languages, as discussed above, 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. This issue is important, because CER patterns often employ the notion of *selection strategies*, which allows them to skip events considered as irrelevant.

■ **Example 9.8** As an example, consider *SREM* (9.3), which may be viewed as a pattern employing the so-called *skip-till-any* selection strategy. The terminal expressions $(\top)^*$ may be considered as such "skipping" operations. This expression tells us that we are interested in T events followed by H events with the same *id*, but we do not really care what happens before the T event or between the T and H events. Therefore, if we want to report the input events contributing to the detection of complex events defined by this *SREM*, then we would need to report only the T and H events with the same *id*, but not any intervening events. Assume that we use this *SREM* to detect complex events on the stream of Table 8.1 and that we have reached the fourth event. In this case, we would need to report as a match the events with indices 2 and 4, but omit event 3. Yet another match would be the one composed of events 1 and 4, ignoring 2 and 3. ■

This flexibility of CER patterns in terms of what input events should be reported introduces additional intricacies that need to be taken into account. For more details about this kind of complexity, please consult [66, 67], where a model for evaluating efficiency in CER is presented, along with complexity results for unary (without registers) symbolic transducers. In the future, we intend to investigate the complexity of decision problems for *SRA* from this point of view.

10. Forecasting with symbolic register automata

We now show how we can use the framework of *SREM* and *SRA* to perform Complex Event Forecasting.

10.1 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 $\mathscr{U}$, we can always create an automaton A_U with states $A.Q \times \mathscr{U}$ 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.

10.1.1 *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 [10]. 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. For details, see Chapter 6.

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 [122]. 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 $\rho = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}, v_{k+1}]$ 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 $\rho' = [1, q_1, v_1] \xrightarrow{\delta_1} [2, q_2, v_2] \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} [k+1, q_{k+1}, v_{k+1}]$, 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 [10].

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 δ_i from the initial run ρ corresponds to a transition δ_i from run ρ'. Since each transition from ρ' 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) [24, 37, 120, 121, 140]. 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 Σ 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 $l(\hat{P}, S_{1..k}) = -\frac{1}{T} \sum_{i=1}^{k} log \hat{P}(t_i \mid$

$t_1 \cdots t_{i-1})$. Minimizing the log-loss is equivalent to maximizing the likelihood $\hat{P}(S_{1..k}) = \prod_{i=1}^{k} \hat{P}(t_i \mid t_1 \dots t_{i-1})$. The average log-loss may also be viewed as a measure of the average compression rate achieved on the test sequence [24]. 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.

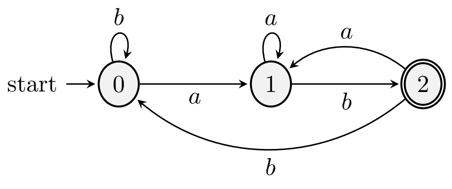
For fixed-order Markov models, the predictor $\hat{P}$ is derived through the estimation of conditional distributions $\hat{P}(\sigma \mid s)$, with m constant and equal to the assumed order of the Markov model (σ is a single symbol and s a string of length m). VMMs, on the other hand, relax the assumption 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 [120, 121], 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 [121] starts with a tree having only a single node, corresponding to the empty string ε. 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 σ 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 [10].

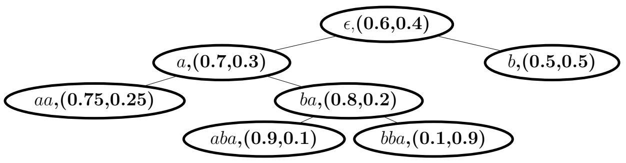
Our goal is to use the *PST* in order to to calculate the so-called waiting-time distribution for every state q of the automaton A. The waiting-time distribution is the distribution of the index n, given by the waiting-time variable $W_q = inf\{n : Y_0, Y_1, ..., Y_n\}$, where $Y_0 = q$, $Y_i \in A.Q \backslash A.Q_f$ 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 10.1 We provide here the intuition through an example. Figure 10.1a shows an example of a deterministic automaton. Note that we use symbols on the transitions, which, as explained, essentially correspond to conditions. Figure 10.1b shows a *PST* which could be constructed from the automaton of Figure 10.1a 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.

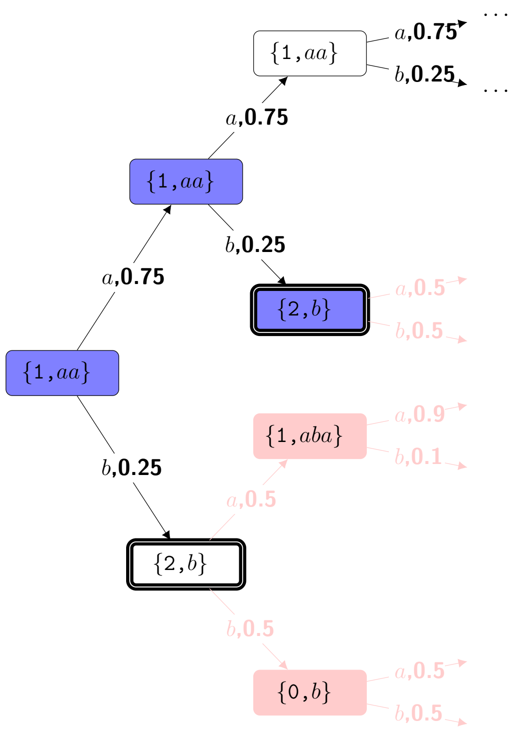
Figure 10.1c illustrates how we can estimate the probability for any future sequence of states of the *dSRA A* of Figure 10.1a, using the distributions of the *PST T* of Figure 10.1b and thus how we can calculate the waiting-time distributions. We first assume that, as the system processes events from 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 10.1a and T has reached its



(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 10.1: Example of estimating waiting-time distributions.

left-most node, $aa, (0.75, 0.25)$ in Figure 10.1b. This is shown as the left-most node also in Figure 10.1c. 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 10.1c, 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 10.1c 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 10.1a 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 10.1c, 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 on 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.

Note that the domain of a waiting-time distribution is not composed of timepoints and thus a forecast, as described previously, does not explicitly refer to time. Each value of the index n on the x axis essentially refers to the number of transitions that the automaton needs to take before reaching a final state, or, equivalently, to the number of future input events to be consumed. In order to produce forecasts referring to time, we need to convert these basic event-related forecasts to time-related ones. This can be achieved by modeling the time elapsed between two successive input events with a (e.g., a Poisson) distribution. We can enrich each node of a *PST* with such distributions. Each *PST* node would thus provide both the probability of encountering a given symbol as the next input element and a distribution giving us the probability for the time it will elapse until this new element arrives.

■ **Example 10.2** For example, consider node $aa, (0.75, 0.25)$ in Figure 10.1b. It informs us that the probability of encountering a, given that the last two symbols are aa, is 0.75. By gathering data from the training dataset about when this a arrives after aa, we can fit a Poisson distribution P_a. We can do the same for the case of b and fit another Poisson distribution, P_b. Now, assume that P_a tells us that a will arrive within an interval of $[0, 3]$ seconds after aa with probability 90% and that b will arrive within an interval of $[2, 4]$ seconds with probability 80%. As a result, for the path with purple nodes in Figure 10.1c, we can infer that we will reach a final state from $\{1, aa\}$ in 2 events and the probability of this happening within an interval of $[0,3]+[2,4]=[2,7]$ seconds is $0.9 \cdot 0.8 = 0.71$, assuming that arrival events are independent. ■

10.1.2 *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 (its minterms, to be more precise) 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 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 10.3** For example, assume that the values for humidity and temperature take only discrete values (low, high) and that we only have two sensors. Then

$$\begin{aligned}\mathcal{U} = \{&(T,1,low),(T,1,high),(T,2,low),(T,2,high),\\ &(H,1,low),(H,1,high),(H,2,low),(H,2,high)\}\end{aligned}$$

We can then map $(T,1,low)$ to a, $(T,1,high)$ 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 $EqualId(\sim,r_1)$, then we do not need 8 symbols. We can use only a and b, with a corresponding to $EqualId(\sim,r_1)$ and b to $\neg EqualId(\sim,r_1)$. 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*. ■

10.1.3 *SREM* with n-ary conditions on an 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.

It must be noted though that such a deterministic automaton cannot work on an unbounded stream as it is. Since it is a windowed automaton, it can work on "streams" whose length is at most that of the window. In other words, we cannot construct a *SRA* that is both fully deterministic and streaming. Even if we use the technique described in Section 9.3 (of adding a new start state and a self-loop transition on it), the resulting automaton will be, as a whole, non-deterministic (the part of the automaton without the new start state will still be deterministic). A possible determinization of the whole new automaton is

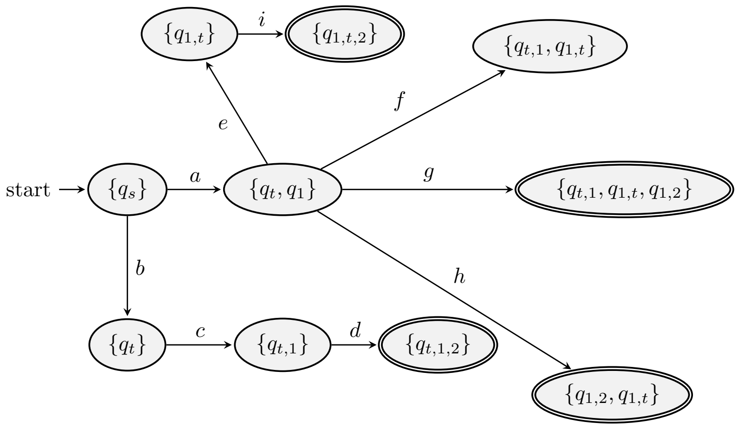


Figure 10.2: Classical deterministic automaton corresponding to Expression (8.2).

not feasible, as we have proven, since this new automaton will no longer be windowed. Instead, we can use the deterministic, windowed *SRA* in a slightly different manner. We can use an actual sliding window of input events, equal in length to the window of the *SRA*, and feed this window of events into the automaton. An issue with this solution is that each input event might belong to different windows and thus might produce different symbols for different windows. We can overcome this problem by creating "hyper-symbols", as sets of "sub-symbols" from each window.

■ **Example 10.4** As an example, Figure 10.2 shows the deterministic classical automaton that can be constructed for *SREM* (8.2) and the *dSRA* of Figure 9.4c (note that Figure 9.4c and thus Figure 10.2 do not depict automata in their totality, but only part of them). For this automaton, we have $w = 3$. Assume that we use the stream of Table 8.1 as a training dataset. We initially feed the first 3 input events it to the automaton of Figure 10.2. 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. The automaton has now reached a "terminal" state with no outgoing transitions. Thus, the third input event does not trigger any transitions. We could map this "invisible" transition to a dummy symbol (e.g., to z), specifically reserved for such situations. We subsequently move the window one step ahead and repeat the same process. The second event of the stream, $(T, 1, 24)$, will now be the first one given to the automaton. Transition a will again be triggered. We repeat for the other two events of the window. Through this process, the event $(T, 1, 24)$ has generated both f (from the first sliding window) and a (from the second window). This combination of (f, a) could be mapped to a "hyper-symbol", say A. We repeat this process until the whole stream of events has been mapped to a stream of hyper-symbols, e.g., $S = A, F, \cdots$. S may now be used to learn a *PST* of a given maximum order. This *PST*, along with the automaton of Figure 10.2, can be used to estimate the waiting-time distributions, as in the example of Figure 10.1c.

■

10.2 Summary

In the last three chapters we presented an automaton model, *SRA*, that can act as a computational model for patterns with n-ary conditions, 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.

IV

Part Four

11. Conclusions & future work

The goal of this thesis has been to develop a formal framework for Complex Event Forecasting (CEF) that has clear semantics, both at the language level (i.e., with respect to the definition of patterns) and at the model level (i.e., with respect to the probabilistic description of the said patterns). Towards this direction, we have presented such a formal framework for CEF, based on various versions of automata (classical, symbolic, symbolic with registers) and Markov models (fixed- and variable-order). Our framework has formal, compositional semantics.

In order to achieve our goal, we had to work in parallel along two axes. First, due to the general lack of computational models (especially automata-based) for Complex Event Recognition (CER) satisfying the requirement for clear, declarative semantics, we incrementally built a suitable framework. We started by showing how classical automata (which are well-studied and do have nice theoretical properties) may be used for CER and CEF. We subsequently moved to a higher level, showing how symbolic automata allow us to define more expressive patterns, while still being suitable for CEF. Finally, we introduced a new computational model, called symbolic register automata, which can capture most of the patterns typically used in CER.

The second axis of our work concerned the construction of appropriate probabilistic models which can describe the behavior of the automaton models we used as our basis. First, we showed how fixed- (or full-) Markov models may be used in a way that allows us to infer how an automaton may behave in the future, given its current state. We demonstrated how such Markov models can be combined with classical and symbolic automata. The next step was to employ variable-order Markov models, so that we can look deeper into the past (when and if this is needed) and make more accurate forecasts. We showed (both theoretically and experimentally) how these Markov models can be combined with symbolic automata. We also showed (theoretically) how and under which conditions they can be combined with symbolic register automata.

Our main contributions thus lie both in conceptualizing and building a CEF system and in proposing an automaton model for CER/F with formal properties that can be systematically evaluated. In most of its versions (classical/symbolic and fixed-/variable-order), our system has been implemented in our open-source engine, Wayeb[1]. To the best of our knowledge, this is the first such framework to be presented in the relevant literature.

There are several directions for future work. One limitation of our framework is that the user still needs to set up the model and there seems to be room for further automation. In particular, the user needs to set the maximum order allowed by the probabilistic model. Additionally, we have started investigating ways to handle concept drift by continuously training and updating the probabilistic model of a pattern. Once we drop the stationarity assumption, it becomes imperative to ensure that any online model updates must be performed very efficiently. Another direction would involve dropping the assumption that patterns are given in advance. Users may know which events they need to predict, without being able to define a pattern for them. Therefore, starting from these target events, pattern mining techniques should be incorporated in order to automatically extract predictive patterns.

In addition to detecting when a CE occurs, we also often need to know which of the input events take part into a detected CE. This functionality would require the use of symbolic transducers in order to mark the input events according to whether they belong to a match or not. We currently do not pay particular attention to this requirement, since our main goal is to forecast the occurrence of CEs, without needing to report the input events that lead to a CE occurrence. We aim to address this requirement in the future.

Our framework could also be used for a task that is not directly related to Complex Event Forecasting. Since predictive modeling and compression are two sides of the same coin, our framework could be used for pattern-driven lossless stream compression, in order to minimize the communication cost, which is a severe bottleneck for geo-distributed CER. The probabilistic model that we construct with our approach could be pushed down to the event sources, such as the vessels in the maritime domain, in order to compress each individual stream and then these compressed streams could be transmitted to a centralized CER engine to perform recognition.

Another direction for future research concerns the effect of using more flexible selection strategies (like *skip-till-any-match*). The usual way to deal with them is to clone runs of the automaton online, when appropriate. We could have followed a similar cloning approach as well and produce forecasts for each run. However, it is doubtful whether individual forecasts made by a multitude (possibly hundreds) of concurrently existing runs would be useful to a user. Some form of aggregate forecasting (e.g., number of full matches expected within the next N events) could be more informative. We intend to pursue this line of research.

We would also like to investigate whether more precise complexity results may be provided for *SRA* or at least for a sub-class of them (e.g., for windowed *SRA*). It is also important to investigate the relationship between *SRA* and other similar automaton models, like automata in sets with atoms [25], Quantified Event Automata [23] and extended symbolic automata [43].

[1]Wayeb source code: `https://github.com/ElAlev/Wayeb`

V

Part Five

A. Appendix for prediction suffix trees

A.1 Complete proof of Proposition 6.1.1

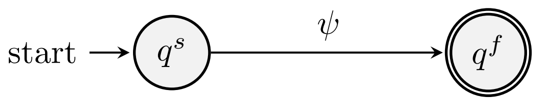
Proposition For every symbolic regular expression R there exists a symbolic finite automaton M such that $\mathscr{L}(R) = \mathscr{L}(M)$.

Proof. Except for the first case, for the other three cases the induction hypothesis is that the theorem holds for the sub-expressions of the initial expression.

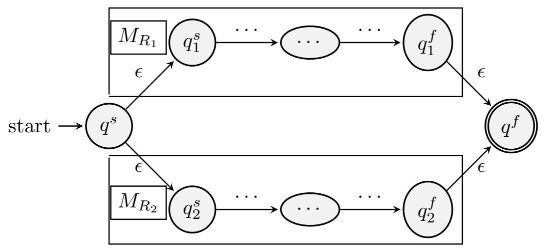
Case where $R = \psi, \psi \in \Psi$.
We construct a SFA as in Figure A.1a. If $w \in \mathscr{L}(R)$, then w is a single character and $w \in [\![\psi]\!]$, i.e., ψ evaluates to TRUE for w. Thus, upon seeing w, the SFA of Figure A.1a moves from q^s to q^f and since q^f is a final state, then w is accepted by this SFA. Conversely, if a string w is accepted by this SFA then it must again be a single character and ψ must evaluate to TRUE since the SFA moved to its final state through the transition equipped with ψ. Thus, $w \in [\![\psi]\!]$ and $w \in \mathscr{L}(R)$.

Case where $R = R_1 + R_2$.
We construct a SFA as in Figure A.1b. If $w \in \mathscr{L}(R)$, then either $w \in \mathscr{L}(R_1)$ or $w \in \mathscr{L}(R_2)$ (or both). Without loss of generality, assume $w \in \mathscr{L}(R_1)$. From the induction hypothesis, it also holds that $w \in \mathscr{L}(M_{R_1})$. Thus, from Figure A.1b, upon reading w, M_{R_1} will have reached q_1^f. Therefore, M_R will have reached q^f throught the ε-transition connecting q_1^f to q^f and thus w is accepted by M_R. Conversely, if $w \in \mathscr{L}(M_R)$, then the SFA M_R of Figure A.1b must have reached q^f and therefore also q_1^f or q_2^f (or both). Assume it has reached q_1^f. Then $w \in \mathscr{L}(M_{R_1})$ and, from the induction hypothesis $w \in \mathscr{L}(R_1)$. Similarly, if its has reached q_2^f, then $w \in \mathscr{L}(R_2)$. Therefore, $w \in \mathscr{L}(R_1) \cup \mathscr{L}(R_2) = \mathscr{L}(R)$.

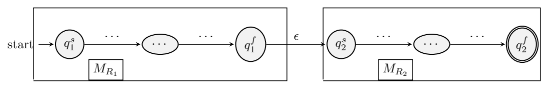
Case where $R = R_1 \cdot R_2$.
We construct a SFA as in Figure A.1c. If $w \in \mathscr{L}(R)$, then $w \in \mathscr{L}(R_1) \cdot \mathscr{L}(R_2)$ or $w = w_1 \cdot w_2$ such that $w_1 \in \mathscr{L}(R_1)$ and $w_2 \in \mathscr{L}(R_2)$. Therefore, from the induction hypothesis, upon reading w_1, M_R will have reached q_1^f and q_2^s. Upon reading the rest of w (w_2),



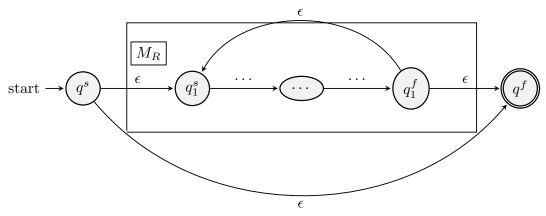
(a) Base case of a single predicate. $R = \psi$.



(b) OR. $R = R_1 + R_2$.



(c) Concatenation. $R = R_1 \cdot R_2$.



(d) Iteration. $R^{'} = R^*$.

Figure A.1: The four cases for constructing a SFA from a SRE

again from the induction hypothesis, M_R will have reached q_2^f. As a result, $w \in \mathscr{L}(M_R)$. Conversely, if $w \in \mathscr{L}(M_R)$, M_R will have reached q_2^f upon reading w and therefore will have also passed through q_1^f upon reading a prefix w_1 of w. Thus, $w = w_1 \cdot w_2$ with $w_1 \in \mathscr{L}(M_{R_1})$ and $w_2 \in \mathscr{L}(M_{R_2})$. From the induction hypothesis, it also holds that $w_1 \in \mathscr{L}(R_1)$ and $w_2 \in \mathscr{L}(R_2)$ and therefore that $w \in \mathscr{L}(R)$.

Case where $R^{'} = R^*$.
We construct a SFA as in Figure A.1d. If $w \in \mathscr{L}(R^{'})$, then $w \in (\mathscr{L}(R))^*$ or, equivalently, $w = w_1 \cdot w_2 \cdot \dots \cdot w_k$ such that $w_i \in \mathscr{L}(R)$ for all w_i. From the induction hypothesis and Figure A.1d, upon reading w_1, $M_{R'}$ will have reached q_1^f and q_1^s. Therefore, the same will be true after reading w_2 and all other w_i, including w_k. Thus, $w \in \mathscr{L}(M_{R'})$. Note that if $w = \varepsilon$, the ε-transition from q^s to q^f ensures that $w \in \mathscr{L}(M_{R'})$. Conversely, assume $w \in \mathscr{L}(M_{R'})$. If $w = \varepsilon$, then by the definition of the * operator, $w \in (\mathscr{L}(\mathscr{R}))^*$. In every other case, $M_{R'}$ must have reached q_1^f and must have passed through q_1^s. Therefore, w may be written as $w = w_1 \cdot w_2$ where $w_2 \in \mathscr{M}_{\mathscr{R}}$ and, for w_1, upon reading it, $M_{R'}$ must have reached q_1^s. There are two cases then: either $w_1 = \varepsilon$ and q_1^s was reached from q^s or $w_1 \neq \varepsilon$ and q_1^s was reached from q_1^f. In the former case, $w = \varepsilon \cdot w_2 = w_2$ and thus $w \in (\mathscr{L}(R))^*$. In the latter case, we can apply a similar reasoning recursively to w_1 in order to split it to sub-strings w_i such that $w_i \in \mathscr{L}(R)$. Therefore, $w \in (\mathscr{L}(R))^*$ and $w \in \mathscr{L}(R^{'})$.

■

A.2 Proof of Proposition 6.1.2

Proposition If $S = t_1, t_2, \cdots$ is a stream of domain elements from an effective Boolean algebra $\mathscr{A} = (\mathscr{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, where $t_i \in \mathscr{D}$, and R is a symbolic regular expression over the same algebra, then, for every $S_{m..k}$, $S_{m..k} \in \mathscr{L}(R)$ iff $S_{1..k} \in \mathscr{L}(R_s)$ (and $S_{1..k} \in \mathscr{L}(M_{R_s})$).

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

A.3 Proof of Theorem 6.2.4

Theorem Let Π be the transition probability matrix of a homogeneous Markov chain Y_t in the form of Equation (4.1) and ξ_{init} its initial state distribution. The probability for the time index n when the system first enters the set of states F, starting from a state in F, can be obtained from

$$P(Y_n \in F, Y_{n-1} \notin F, \cdots, Y_1 \in F \mid \xi_{init}) = \begin{cases} \xi_F{}^T F 1 & \text{if } n = 2 \\ \xi_F{}^T F_N N^{n-2}(I-N)1 & \text{otherwise} \end{cases}$$

where ξ_F is the vector consisting of the elements of ξ_{init} corresponding to the states of F.

Proof. **Case where $n = 2$.**
In this case, we are in a state $i \in F$ and we take a transition that leads us back to F again. Therefore, $P(Y_2 \in F, Y_1 = i \in F \mid \xi_{init}) = \xi(i) \sum_{j \in F} \pi_{ij}$, i.e., we first take the probability of starting in i and multiply it by the sum of all transitions from i that lead us back to F. This result folds for a certain state $i \in F$. If we start in any state of F, $P(Y_2 \in F, Y_1 \in F \mid \xi_{init}) = \sum_{i \in F} \xi(i) \sum_{j \in F} \pi_{ij}$. In matrix notation, this is equivalent to $P(Y_2 \in F, Y_1 \in F \mid \xi_{init}) = \xi_F{}^T F 1$.

Case where $n > 2$.
In this case, we must necessarily first take a transition from $i \in F$ to $j \in N$, then, for multiple transitions we remain in N and we finally take a last transition from N to F. We can write

$$\begin{aligned} P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \in F \mid \xi_{init}) =& P(Y_n \in F, Y_{n-1} \notin F, ..., Y_2 \notin F \mid \xi_N^{'}) \\ =& P(Y_{n-1} \in F, Y_{n-2} \notin F, ..., Y_1 \notin F \mid \xi_N^{'}) \end{aligned} \tag{A.1}$$

where $\xi_N^{'}$ is the state distribution (on states of N) after having taken the first transition from F to N. This is given by $\xi_N^{'} = \xi_F{}^T F_N$. By using this as an initial state distribution in Eq. 6.2 and running the index n from 1 to $n-1$, as in Eq. A.1, we get

$$P(Y_n \in F, Y_{n-1} \notin F, ..., Y_1 \in F \mid \xi_{init}) = \xi_F{}^T F_N N^{n-2}(I-N)1$$

■

A.4 Proof of correctness for Algorithm 3

Proposition Let P be a waiting-time distribution with horizon h and let $\theta_{fc} < 1.0$ be a confidence threshold. Algorithm 3 correctly finds the smallest interval whose probability exceeds θ_{fc}.

Proof of correctness for Algorithm 3. We only need to prove the loop invariant. Assume that after the k^{th} iteration of the outer while loop $i = i_k$ and $j = j_k$ and that after the

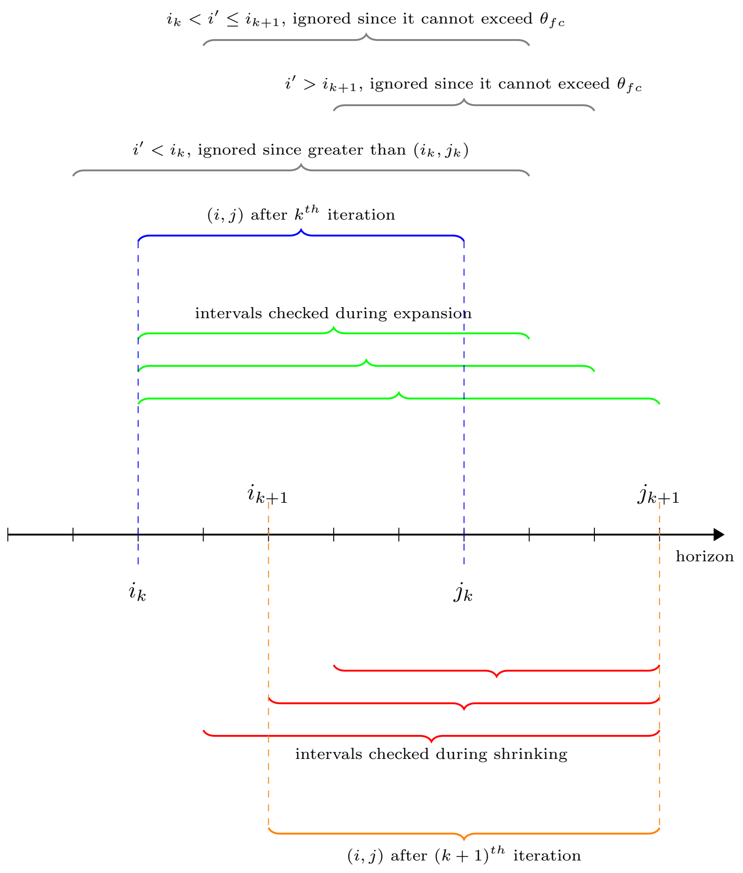

$i_k < i' \leq i_{k+1}$, ignored since it cannot exceed θ_{fc}
$i' > i_{k+1}$, ignored since it cannot exceed θ_{fc}
$i' < i_k$, ignored since greater than (i_k, j_k)
(i, j) after k^{th} iteration
intervals checked during expansion
i_{k+1}
j_{k+1}
horizon
i_k
j_k
intervals checked during shrinking
(i, j) after $(k+1)^{th}$ iteration

Figure A.2: One iteration of Algorithm 3. After the k^{th} iteration, (i, j) is either the smallest interval or there already exists a smaller one. In the $(k+1)^{th}$ iteration, we need to check intervals for which $j_k < j' \leq j_{k+1}$, .i.e., whose right limit is greater than j_k and smaller (but including) j_{k+1}. Out of those intervals, the ones that start before i_k need not be checked because they are greater than (i_k, j_k). The ones that start after i_{k+1} are also ignored, since we already know, from the shrinking phase, that their super-interval $(i_{k+1}+1, j_{k+1})$ does not exceed θ_{fc}. Finally, we can also ignore the intervals that start between i_k and i_{k+1} (not including i_k) and end before j_{k+1} (not including j_{k+1}), because they cannot exceed θ_{fc}. If such an interval actually existed, then the expansion phase would have stopped before j_{k+1}.

$(k+1)^{th}$ iteration $i = i_{k+1}$ and $j = j_{k+1}$. If the invariant holds after the k_{th} iteration, then all intervals with $e \leq j_k$ have been checked and we know that (s, e) is the best interval up to j_k. It can be shown that, during the $(k+1)^{th}$ iteration, the intervals up to j_{k+1} that are not explicitly checked are intervals which cannot possibly exceed θ_{fc} or cannot be better than the currently held best interval (s, e). There are three such sets of unchecked intervals (see also Figure A.2):

- All intervals (i', j') such that $i' < i_k$ and $j_k \leq j' \leq j_{k+1}$, i.e., we ignore all intervals that start before i_k. Even if these intervals exceed θ_{fc}, they cannot possibly be smaller than (s, e), since we know that $(s, e) = (i_k, j_k)$ or that (s, e) is even smaller than (i_k, j_k).
- All intervals (i', j') such that $i' > i_{k+1}$ and $j_k \leq j' \leq j_{k+1}$, i.e., we ignore all intervals that start after i_{k+1}. These intervals cannot possibly exceed θ_{fc}, since $(i_{k+1}+1, j_{k+1})$ is below θ_{fc} and all these intervals are sub-intervals of $(i_{k+1}+1, j_{k+1})$.
- We are thus left with intervals (i', j') such that $i_k \leq i' \leq i_{k+1}$ and $j_k \leq j' \leq j_{k+1}$.

Out of all the interval that can be constructed from combining these i' and j', the algorithm checks the intervals $(i' = i_k, j')$ and $(i', j' = j_{k+1})$. The intervals that are thus left unchecked are the intervals (i', j') such that $i_k < i' \leq i_{k+1}$ and $j_k \leq j' < j_{k+1}$. The question is: is it possible for such an interval to exceed θ_{fc}. The answer is negative. Assume that there is such an interval (i', j'). If this were the case, then the algorithm, during its expansion phase, would have stopped at j', because (i_k, j') would exceed θ_{fc}. Therefore, these intervals cannot exceed θ_{fc}.

A similar reasoning allows us to show that the loop invariant holds after the first iteration. It thus holds after every iteration. ■

A.5 Proofs of complexity results

A.5.1 Proof of Proposition 6.3.1

Proposition **— Step 1 in Figure 6.9.** Let $S_{1..k}$ be a stream and m the maximum depth of the Counter Suffix Tree T to be constructed from $S_{1..k}$. The complexity of constructing T is $O(m(k-m))$.

Proof. There are three operations that affect the cost: incrementing the counter of a node by 1, with constant cost i; inserting a new node, with constant cost n; visiting an existing node with constant cost v; We assume that $n > v$. For every $S_{l-m+1..l}$, $m \leq l \leq k$ of length m, there will be m increment operations and m nodes will be "touched", i.e., either visited if already existing or created. Therefore, the total number of increment operations is $(k-m+1)m = km - m^2 + m = m(k-m) + m$. The same result applies for the number of node "touches". It is always true that $m < k$ and typically $m \ll k$. Therefore, the cost of increments is $O(m(k-m))$ and the cost of visits/creations is also $O(m(k-m))$. Thus, the total cost is $O(m(k-m)) + O(m(k-m)) = O(m(k-m))$. In fact, the worst case is when all $S_{l-m+1..l}$ are different and have no common suffixes. In this case, there are no visits to existing nodes, but only insertions, which are more expensive than visits. Their cost would again be $O(nm(k-m)) = O(m(k-m))$, ignoring the constant n. ■

A.5.2 Proof of Proposition 6.3.2

Proposition **— Step 3a in Figure 6.9.** Let T be a *PST* of maximum depth m, learned with the t minterms of a *DSFA* M_R. The complexity of constructing a *PSA* M_S from T is $O(t^{m+1} \cdot m)$.

Proof. We assume that the cost of creating new states and transitions for M_S is constant. In the worst case, all possible suffixes of length m have to be added to T as leaves. T will thus have t^m leaves. The main idea of the algorithm for converting a *PST* T to a *PSA* M_S is to use the leaves of T as states of M_S and for every symbol (minterm) σ find the next state/leaf and set the transition probability to be equal to the probability of σ from the source leaf. If we assume that the cost of accessing a leaf is constant (e.g., by keeping separate pointers to the leaves), the cost for constructing M_S is dominated by the cost of constructing the k^m states of M_S and the t transitions from each such state. For each transition, finding the next state requires traversing a path of length m in T. The total cost is thus $O(t^m \cdot t \cdot m) = O(t^{m+1} \cdot m)$. ■

A.5.3 Proof of Proposition 6.3.4

Proposition — Step 4 in Figure 6.9. Let M_R be a *DSFA* with t minterms and M_S a *PSA* learned with the minterms of M_R. The complexity of constructing an embedding M of M_S in M_S with Algorithm 2 is $O(t \cdot |M_R.Q \times M_S.Q|)$.

Proof. We assume that the cost of constructing new states and transitions for M is constant. We also assume that the cost of finding a given state in both M_R and M_S is constant, e.g., by using a linked data structure for representing the automaton with a hash table on its states (or an array), and the cost of finding the next state from a given state is also constant. In the worst case, even with the incremental algorithm 2, we would need to create the full Cartesian product $M_R.Q \times M_S.Q$ to get the states of M. For each of these states, we would need to find the states of M_R and M_S from which it will be composed and to create t outgoing transitions. Therefore, the complexity of creating M would be $O(t \cdot |M_R.Q \times M_S.Q|)$. ■

A.5.4 Proof of Proposition 6.3.5

Proposition — Step 5 in Figure 6.9. Let M be the embedding of a *PSA* M_S in a *DSFA* M_R. The complexity of estimating the waiting-time distribution for a state of M and a horizon of length h using Theorem 6.2.3 is $O((h-1)k^{2.37})$ where k is the dimension of the square matrix N.

Proof. We want to use Equation 6.2 to estimate the distribution of a state. The equation is repeated below:

$$P(Y_n \in F, Y_{n-1} \notin F, \dots, Y_1 \notin F \mid \xi_{init}) = {\xi_N}^T N^{n-1}(I-N)\mathbf{1}$$

We want to estimate the distribution for the h points of the horizon, i.e., for $n=2$, $n=3$ up to $n=h+1$. For $n=2$, we have

$$P(Y_2 \in F, Y_1 \notin F \mid \xi_{init}) = {\xi_N}^T N(I-N)\mathbf{1}$$

For $n=3$, we have

$$P(Y_3 \in F, Y_2 \notin F, Y_1 \notin F \mid \xi_{init}) = {\xi_N}^T N^2(I-N)\mathbf{1}$$

In general, for $n=i$, we can use the power of N that we have estimated in the previous step for $n=i-1$, N^{i-2}, in order to estimate the next power N^{i-1} via a multiplication by N so as to avoid estimating this power from scratch. Then N^{i-1} can be multiplied by $(I-N)\mathbf{1}$, which remains fixed for all i and can thus be calculated only once.

The cost of estimating $(I-N)$ is k^2 due to the k^2 subtractions. Multiplying the matrix $(I-N)$ by the vector $\mathbf{1}$ results in a new vector with k elements. Each of these elements requires k multiplications and $k-1$ additions or $2k-1$ operations. Thus, the estimation of $(I-N)\mathbf{1}$ has a cost of $k^2+k(2k-1)=3k^2-k$.

Now, for $n=i$, estimating the power N^{i-1} from N^{i-2} has a cost of $k^{2.37}$ using an efficient multiplication algorithm such as the Coppersmith–Winograd algorithm [38] or the improvement proposed by Stothers [130].

Additionally, N^{i-1} must then be multiplied by the vector $(I-N)\mathbf{1}$, with a cost of $2k^2-k$, resulting in a new vector with k elements.

This vector must then be multiplied by ${\xi_N}^T$ to produce the final probability value with a cost of $2k-1$ for the k multiplications and the $k-1$ additions.

We thus have a fixed initial cost of $3k^2 - k$ and then for every iteration i a cost of $k^{2.37}I_{\{i>2\}} + 2k^2 - k + 2k - 1 = k^{2.37}I_{\{i>2\}} + 2k^2 + k - 1$, where $I_{\{i>2\}}$ is an indicator function (1 for $i > 2$ and 0 otherwise). Note that the cost $k^{2.37}$ is not included for $i = 2$ because in this case we do not need to raise N to a power. The total cost would thus be:

$$\begin{aligned}
3k^2 - k + & k^{2.37} \cdot 0 + 2k^2 + k - 1 && \text{for } n = 2 \\
+ & k^{2.37} \cdot 1 + 2k^2 + k - 1 && \text{for } n = 3 \\
& \dots \\
+ & k^{2.37} \cdot 1 + 2k^2 + k - 1 && \text{for } n = h + 1 \\
= & (h-1)k^{2.37} + (2h+3)k^2 + (h-1)k - h && = O((h-1)k^{2.37})
\end{aligned}$$

■

A.5.5 Proof of Proposition 6.3.6

Proposition — Step 6 in Figure 6.9. For a waiting-time distribution with a horizon of length h, the complexity of finding the smallest interval that exceeds a confidence threshold θ_{fc} with Algorithm 3 is $O(h)$.

Proof. Indexes i and j of Algorithm 3 scan the distribution only once. The cost for j is the cost of h points of the distribution that need to be accessed plus $h - 1$ additions. Similarly, the cost for i is the cost of (at most) h accessed points plus the cost of (at most) $h - 1$ subtractions. Thus the total cost is $O(h)$. ■

A.5.6 Proof of Proposition 6.3.3

Proposition — Step 3b in Figure 6.9. Let T be a *PST* of maximum depth m, learned with the t minterms of a *DSFA* M_R. The complexity of estimating the waiting-time distribution for a state of M_R and a horizon of length h directly from T is $O((m+3)\frac{t-t^{h+1}}{1-t})$.

Proof. After every new event arrival, we first have to construct the tree of future states, as shown in Figure 6.7b. In the worst case, no paths can be pruned and the tree has to be expanded until level h. The total number of nodes that have to be created is thus a geometric progress: $t + t^2 + \cdots + t^h = \sum_{i=1}^{h} t^i = \frac{t-t^{h+1}}{1-t}$. Assuming that it takes constant time to create a new node, this formula gives the cost of creating the nodes of the trees. Another cost that is involved concerns the time required to find the proper leaf of the *PST* T before the creation of each new node. In the worst case, all leaves will be at level m. The cost of each search will thus be m. The total search cost for all nodes will be $mt + mt^2 + \cdots + mt^h = \sum_{i=1}^{h} mt^i = m\frac{t-t^{h+1}}{1-t}$. The total cost (node creation and search) for constructing the tree is $\frac{t-t^{h+1}}{1-t} + m\frac{t-t^{h+1}}{1-t} = (m+1)\frac{t-t^{h+1}}{1-t}$. With the tree of future states at hand, we can now estimate the waiting-time distribution. In the worst case, the tree will be fully expanded and we will have to access all its paths until level h. We will first have to visit the t nodes of level 1, then the t^2 nodes of level 2, etc. The access cost will thus be $t + t^2 + \cdots + t^h = \sum_{i=1}^{h} t^i = \frac{t-t^{h+1}}{1-t}$. We also need to take into account the cost of estimating the probability of each node. For each node, one multiplication is required, assuming that we store partial products and do not have to traverse the whole path to a node to estimate its probability. As a result, the number of multiplications will also be $\frac{t-t^{h+1}}{1-t}$. The total cost (path traversal and multiplications) will thus be $2\frac{t-t^{h+1}}{1-t}$, where we ignore the cost of summing the probabilities of final states, assuming it is constant. By adding the cost of constructing the tree ($(m+1)\frac{t-t^{h+1}}{1-t}$) and the cost of estimating the distribution ($2\frac{t-t^{h+1}}{1-t}$), we get a complexity of $O((m+3)\frac{t-t^{h+1}}{1-t})$. ■

B. Appendix for SREM

B.1 Proof of Theorem 9.2.1

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

Proof. For a *SREM* e and valuations v, v', let $\mathscr{L}(e,v,v')$ denote all strings S such that $(e,S,v) \vdash v'$. Similarly, for a *SRA* A, let $\mathscr{L}(A,v,v')$ 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 $\mathscr{L}(e) = \mathscr{L}(A)$ or that $\mathscr{L}(e,v,v') = \mathscr{L}(A,v,v')$. The latter implies that $\mathscr{L}(e,\sharp,v'') = \mathscr{L}(A,\sharp,v'')$ for some valuation v'' or equivalently $\mathscr{L}(e) = \mathscr{L}(A)$, which is our goal. The proof is inductive. We prove directly the base cases for the simple expressions $e := \emptyset$, $e := \varepsilon$, $e := \phi = R(x_1, \cdots, x_n)$ and $e := \phi = R(x_1, \cdots, x_n) \downarrow w$. For the complex expression $e := e_1 \cdot e_2$, $e := e_1 + e_2$ and $e' = e^*$, we use as an inductive hypothesis that our target result hods for the sub-expressions and then prove that it also holds for the top expression. For example, for $e := e_1 \cdot e_2$, we assume that $\mathscr{L}(e_1,v,v'') = \mathscr{L}(A_1,v,v'')$ and that $\mathscr{L}(e_2,v'',v') = \mathscr{L}(A_2,v'',v')$.

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 $reg(e)$ be a function applied to a *SREM* e. We define it as follows:

$$reg(e) = \begin{cases} \emptyset & \text{if } e = \emptyset \\ \emptyset & \text{if } e = \varepsilon \\ \{x_1\} \cup \cdots \cup \{x_n\} \cup \{w\} & \text{if } e = R(x_1, \cdots, x_n) \downarrow w \\ reg(e_1) \cup reg(e_2) & \text{if } e = e_1 \cdot e_2 \\ reg(e_1) \cup reg(e_2) & \text{if } e = e_1 + e_2 \\ reg(e_1) & \text{if } e = (e_1)^* \end{cases} \tag{B.1}$$

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

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

Assume $e := \varepsilon$**.** We know that $\mathscr{L}(e) = \{\varepsilon\}$. We can then construct a *SRA* $A = (Q, q_s, Q_f, R, \Delta)$ where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \varepsilon \downarrow \emptyset \rightarrow q_f$. See Figure B.1a. 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 ε transition. Thus $\mathscr{L}(A) = \{\varepsilon\}$.

Assume $e := \phi = R(x_1, \cdots, x_n)$**, where** ϕ **is a condition and all** x_i **belong to a set of register variables** $\{r_1, \cdots, r_k\}$**.** We construct the following *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 $\delta = q_s, \phi \downarrow \emptyset \rightarrow q_f$. See Figure B.1b.

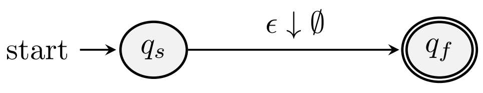
We first prove $S \in \mathscr{L}(e, v, v') \Rightarrow S \in \mathscr{L}(A, v, v')$ for a string S. It is obvious that S must be composed of a single element, i.e., $S = t_1$. Since $S = t_1$ is accepted by e starting from the valuation v, this means that $(\phi, S, v) \vdash v'$, with $v' = v$, according to the second case of Definition 8.2.10. Thus $(t_1, v) \models \phi$. This then implies that the second case in the definition of a successor configuration (see Definition 9.1.2) 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 $S \in \mathscr{L}(A, v, v')$.

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

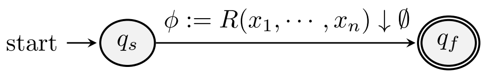
Assume $e := \phi = R(x_1, \cdots, x_n) \downarrow w$**, where** ϕ **is a condition, all** x_i **belong to a set of register variables** $\{r_1, \cdots, r_k\}$ **and** w **a write register (not necessarily one of** r_i**).** We construct the following *SRA* $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = R_{top}$, $\Delta = \{\delta\}$ and $\delta = q_s, \phi \downarrow \{w\} \rightarrow q_f$. See Figure B.1c.

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 9.1.2). 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.

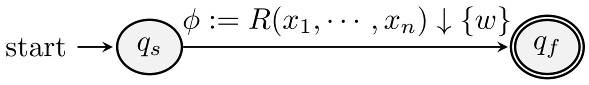
Assume $e := e_1 \cdot e_2$**, where** e_1 **and** e_2 **are** *SREM***.** We first construct A_1 and A_2, the *SRA* for e_1 and e_2 respectively. We construct the following *SRA* $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = A_1.Q \cup A_2.Q$, $q_s = A_1.q_s$, $Q_f = \{A_2.q_f\}$, $R = R_{top}$, $\Delta = A_1.\Delta \cup A_2.\Delta \cup \{\delta\}$ and $\delta = A_1.q_f, \varepsilon \rightarrow A_2.q_s$. See Figure B.1d. We thus simply connect A_1 and A_2 with an ε



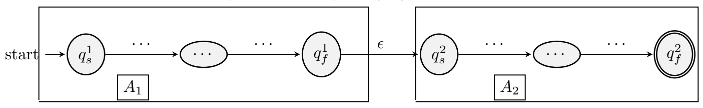
(a) Base case of a single ε condition, $e := \varepsilon$.



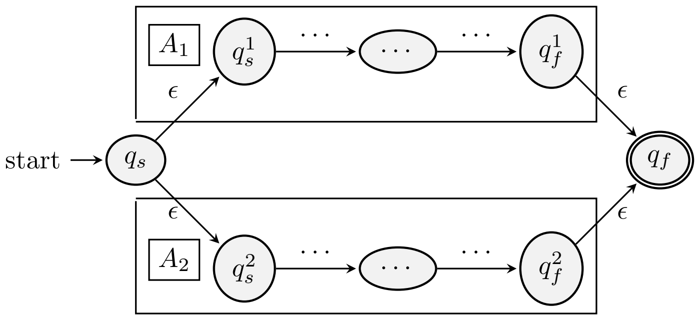
(b) Base case of a single condition, $e := \phi = R(x_1, \cdots, x_n)$.



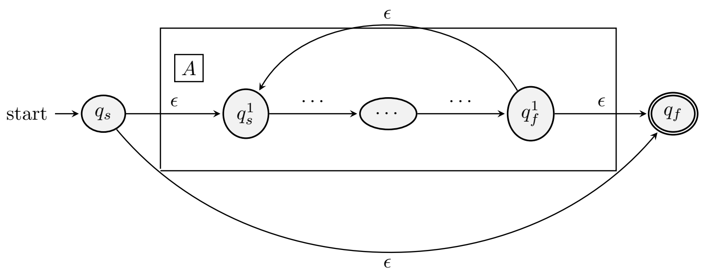
(c) Base case of a single condition with a write register, $e := \phi \downarrow W = R(x_1, \cdots, x_n) \downarrow \{w\}$.



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



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



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

Figure B.1: The cases for constructing a *SRA* from a *SREM*.

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 $S \in \mathscr{L}(e,v,v') \Rightarrow S \in \mathscr{L}(A,v,v')$ for a string S. Since $S \in \mathscr{L}(e,v,v')$, 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 $S_1 \in \mathscr{L}(e_1,v,v'')$ and $S_2 \in \mathscr{L}(e_2,v'',v')$. 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 $S_1 \in \mathscr{L}(A_1,v,v'')$ and $S_2 \in \mathscr{L}(A_2,v'',v')$. 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 $S_1 \in \mathscr{L}(A_1,v,v'')$ it follows that there exists an accepting run ρ_1 of A_1 over S_1 such that $\rho_1 = [1,A_1.q_s,v] \rightarrow \cdots \rightarrow [l_1+1,A_1.q_f,v'']$. Similarly, from $S_2 \in \mathscr{L}(A_2,v'',v')$ it follows that there exists an accepting run ρ_2 of A_2 over S_2 such that $\rho_2 = [1,A_2.q_s,v''] \rightarrow \cdots \rightarrow [l_2+1,A_2.q_f,v']$. Let's construct a run by connecting ρ_1 and ρ_2 with an ε transition: $\rho = [1,A_1.q_s,v] \rightarrow \cdots \rightarrow [l_1+1,A_1.q_f,v''] \overset{A_1.q_f,\varepsilon \rightarrow A_2.q_s}{\rightarrow} [l_1+2,A_2.q_s,v''] \rightarrow \cdots \rightarrow [l_1+l_2+1,A_2.q_f,v']$. We can see that this is indeed an accepting run of A. Thus $S \in \mathscr{L}(A,v,v')$.

The inverse direction, $S \in \mathscr{L}(A,v,v') \Rightarrow S \in \mathscr{L}(e,v,v')$, can be proven in a similar manner. Since $S \in \mathscr{L}(A,v,v')$, there exists an accepting run ρ of A over S. By the construction of A, however, this run must be in the form $\rho = \rho_1 \overset{\varepsilon}{\rightarrow} \rho_2$ with ρ_1 being an accepting run of A_1 over a string S_1 and ρ_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 $S_1 \in \mathscr{L}(e_1,v,v'')$ and $S_2 \in \mathscr{L}(e_2,v'',v')$ and finally that $S \in \mathscr{L}(e,v,v')$.

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 $\delta_{s,1} = q_s,\varepsilon \rightarrow A_1.q_s$, $\delta_{s,2} = q_s,\varepsilon \rightarrow A_2.q_s$, $\delta_{1,f} = A_1.q_f,\varepsilon \rightarrow q_f$, $\delta_{2,f} = A_2.q_f,\varepsilon \rightarrow q_f$. See Figure B.1e. We thus create a new state, q_s, acting as the start state and connect it through ε 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 $S \in \mathscr{L}(e,v,v') \Rightarrow S \in \mathscr{L}(A,v,v')$ 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 B.1f. We first construct 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 $S \in \mathscr{L}(e,v,v') \Rightarrow S \in \mathscr{L}(A,v,v')$ for a string S. Since $S \in \mathscr{L}(e,v,v')$, $S = S_1 \cdot S'$ 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 $\rho = \rho_1 \overset{\varepsilon}{\rightarrow} \rho_2 \overset{\varepsilon}{\rightarrow} \cdots \overset{\varepsilon}{\rightarrow} \rho_n$. It is easy to see that ρ is an accepting run of A'. Similarly for the inverse direction. ■

B.2 Proof of Lemma 9.2.2

Lemma For every *SRA* A_ε with ε transitions there exists an equivalent *SRA* $A_{\not\in}$ without ε transitions, i.e., a *SRA* such that $\mathscr{L}(A_\varepsilon) = \mathscr{L}(A_{\not\in})$.

Algorithm 5: Eliminating ε-transitions (*EliminateEpsilon*).

```
Input: SRA $A_{\varepsilon}$, possibly with $\varepsilon$ transitions
Output: SRA $A_{\notin}$ without $\varepsilon$-transitions
$q_{\notin,s} \leftarrow Enclose(A_{\varepsilon}.q_s)$; $Q_{\notin} \leftarrow \{q_{\notin,s}\}$; $\Delta_{\notin} \leftarrow \emptyset$;
if $\exists q \in q_{\notin,s} : q \in A_{\varepsilon}.Q_f$ then
    $Q_{\notin,f} \leftarrow \{q_{\notin,s}\}$;
else
    $Q_{\notin,f} \leftarrow \emptyset$;
$frontier \leftarrow \{q_{\notin,s}\}$;
while $frontier \neq \emptyset$ do
    $q_{\notin} \leftarrow$ pick an element from $frontier$;
    foreach $q_{\varepsilon} \in q_{\notin}$ do
        foreach $\delta_{\varepsilon} \in A_{\varepsilon}.\Delta : \delta_{\varepsilon}.source = q_{\varepsilon} \wedge \delta_{\varepsilon} \neq \varepsilon$ do
            $p_{\varepsilon} \leftarrow \delta_{\varepsilon}.target$;
            $p_{\notin} \leftarrow Enlose(p_{\varepsilon})$;
            $Q_{\notin} \leftarrow Q_{\notin} \cup \{p_{\notin}\}$;
            if $\exists q \in p_{\notin} : q \in A_{\varepsilon}.Q_f$ then
                $Q_{\notin,f} \leftarrow Q_{\notin,f} \cup \{p_{\notin}\}$;
            $\delta_{\notin} \leftarrow CreateNewTransition(q_{\notin}, \delta_{\varepsilon}.\phi \downarrow \delta_{\varepsilon}.W \rightarrow p_{\notin})$;
            $\Delta_{\notin} \leftarrow \Delta_{\notin} \cup \{\delta_{\notin}\}$;
            $frontier \leftarrow frontier \cup \{p_{\notin}\}$;
    $frontier \leftarrow frontier \setminus \{q_{\notin}\}$;
$A_{\notin} \leftarrow (Q_{\notin}, q_{\notin,s}, Q_{\notin,f}, A_{\varepsilon}.R, \Delta_{\notin})$;
return $A_{\notin}$;
```

Proof. We first give the algorithm. See Algorithm 5. Note that in this algorithm, the function *Enclose* is the usual function for ε-enclosure in standard automata theory and we will not repeat it here (see [76]). Suffice it to say that, when applied to a state q (or set of states $\{q_i\}$), it returns all the states we can reach from q (or all q_i) by following only ε-transitions. It is also worth noting that the algorithm does not create the power-set of states and then connects them through transitions. It creates those subsets it needs by "forward-looking" for what is necessary, but it is equivalent to the power-set construction algorithm. We will prove that $S \in \mathscr{L}(A_{\varepsilon}) \Leftrightarrow S \in \mathscr{L}(A_{\notin})$ for a string S.

We first prove the direction $S \in \mathscr{L}(A_{\varepsilon}) \Rightarrow S \in \mathscr{L}(A_{\notin})$. The other direction can be proven similarly. Let ρ_{ε} denote an accepting run of A_{ε} over S, where $k = |S|$ is the length of S.

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

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

$$\begin{aligned}
\rho_{\notin} = & [1, q_{\notin,1} = q_{\notin,s}, v_{\notin,1} = \sharp] && \text{sub-run 1} \\
& \overset{\delta_{\notin,1}}{\rightarrow} [2, q_{\notin,2}, v_{\notin,2}] && \text{sub-run 2} \\
& \dots \\
& \overset{\delta_{\notin,i-1}}{\rightarrow} [i, q_{\notin,i}, v_{\notin,i}] && \text{sub-run i} \\
& \overset{\delta_{\notin,i}}{\rightarrow} [i+1, q_{\notin,i+1}, v_{\notin,i+1}] && \text{sub-run i+1} \\
& \dots \\
& \overset{\delta_{\notin,k}}{\rightarrow} [k+1, q_{\notin,k+1}, v_{\notin,k+1}] && \text{sub-run k+1}
\end{aligned} \tag{B.3}$$

$\rho^{\notin}$ necessarily follows k transitions, since it does not have any ε-transitions. On the other hand, ρ^{ε} may follow more than k transitions ($j \geq k$), because several ε transitions may intervene between "actual", non-ε transitions, as shown in Run B.2. The number of non-ε transitions is still k. ρ_{ε} is thus necessarily composed of k "sub-runs", where the first configuration of each sub-run is reached via a non-ε transition, followed by a sequence of 0 or more ε transitions. Each line in Run B.2 is such a sub-run. We can also split $\rho_{\notin}$ in sub-runs, but in this case each such sub-run will be simply composed of a single configuration. See Run B.3.

We will prove the following. For each sub-run i of ρ_{ε}, it holds that:

1. $q_{\notin,i} \in q_{\varepsilon,i}$. In fact, $q_{\notin,i} = Enclosure(q_{\varepsilon,i})$.
2. $v_{\varepsilon,i} = v_{\notin,i}$, i.e., A_{ε} and $A_{\notin}$ 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, $q_{\notin,k+1} \in Q_{\notin,f}$ and $\rho_{\notin}$ is an accepting run as well.

First, notice that in each sub-run i of ρ_{ε}, $v_{\varepsilon,i}$ remains the same, since ε transitions never modify the contents of the registers. Thus, in ρ_{ε}, $v_{\varepsilon,i'} = v_{\varepsilon,i}$. It is also obviously true that $i' = i$, since ε transitions do not read elements from S and thus the automaton's head does not move. The only thing that could possibly change is $q_{\varepsilon,i'}$, so that, in general, $q_{\varepsilon,i'} \neq q_{\varepsilon,i}$. Therefore, in Run B.2, we move from sub-run i to sub-run $i+1$ by jumping from $q_{\varepsilon,i'}$ to $q_{\varepsilon,i+1}$. This implies that $\delta_{\phi,i}$, connecting $q_{\varepsilon,i'}$ to $q_{\varepsilon,i+1}$, is triggered when the contents of the register are those of $v_{\varepsilon,i'} = v_{\varepsilon,i}$.

Now, $q_{\varepsilon,i'}$ belongs to the enclosure of $q_{\varepsilon,i}$. Otherwise, it would be impossible to reach it from $q_{\varepsilon,i}$ by following only ε transitions. From the induction hypothesis we know that $q_{\notin,i}$ must be the enclosure of $q_{\varepsilon,i}$. From the construction algorithm for $A_{\notin}$ (Algorithm 5) we also know that the transition $\delta_{\varepsilon,i}$ also exists in $A_{\notin}$, with $q_{\notin,i}$ its source. $\delta_{\notin,i}$ is has the same condition and references the same registers as $\delta_{\varepsilon,i}$. Since $\delta_{\varepsilon,i}$ is triggered with $v_{\varepsilon,i'}$, $\delta_{\notin,i}$ must also be triggered because $v_{\notin} = v_{\varepsilon,i}$ (by the induction hypothesis) and thus $v_{\notin} = v_{\varepsilon,i'}$. From the construction algorithm, we can see that $q_{\notin,i+1}$ will be the enclosure of $q_{\varepsilon,i+1}$ The state q_e in Algorithm 5 is $q_{\varepsilon,i'}$ in Run B.2, while state p_{ε} in the algorithm is state $q_{\varepsilon,i+1}$ in ρ_{ε}. $p_{\notin}$ is the thus the enclosure of $q_{\varepsilon,i+1}$. Thus there exists $q_{\notin,i+1}$ which we can reach from $q_{\notin,i}$ and which is the enclosure of $q_{\varepsilon,i}$. The second part of the induction hypothesis is obviously true for $i+1$, i.e., $v_{\varepsilon,i+1} = v_{\notin,i+1}$, since exactly the same registers are modified in exactly the same way by $\delta_{\varepsilon,i}$ and $\delta_{\notin,i}$.

Therefore, $q_{\varepsilon,k+1} \in q_{\notin,k+1}$ which implies that $q_{\notin,k+1} \in Q_{\notin,f}$ and thus $\rho_{\notin}$ is an accepting run of $A_{\notin}$ over S. ■

B.3 Proof of Lemma 9.2.3

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

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 [79] 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 $p = (p_1, \cdots, p_w) \in (2^{R_{mr}})^w : \bigcup_{k=1}^{w} = R_{mr}$ 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 $P_{R_{mr}} = \{p \mid p \text{ is a partition of } R_{mr}\}$ 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 $p_s = (R_{mr}, \overbrace{\emptyset, \cdots, \emptyset}^{w-1})$.
- $Q_{sr,f} = Q_{mr,f} \times P_{R_{mr}}$.
- $R_{mr} = \{r_{mr,1}, \cdots, r_{mr,w}\}$.
- The set of transitions Δ_{sr} is defined as follows:
  - For every $\delta_{mr} \in \Delta_{mr} : \delta_{mr} = q_{mr}, \varepsilon \rightarrow q'_{mr}$, i.e., for every ε transition of A_{mr}, we add the transitions $\delta_{sr} = (q_{mr}, p), \varepsilon \rightarrow (q'_{mr}, p)$, one for each $p \in P_{R_{mr}}$.
  - For every $\delta_{mr} \in \Delta_{mr} : \delta_{mr} = q_{mr}, \phi_{mr} \rightarrow q'_{mr}$, i.e., for every non-$\varepsilon$ transition of A_{mr} that does not write to any registers, we do the following. Let R_{ϕ} denote the set of registers referenced/accessed by ϕ_{mr}. For every $p \in P_{R_{mr}}$ we add a transition $\delta_{sr} = (q_{mr}, p), \phi_{sr} \rightarrow (q'_{mr}, p)$. ϕ_{sr} is the same as ϕ_{mr} with the following difference. Each $r_{mr,i} \in R_{\phi}$ (i.e., each register referenced by ϕ_{mr}) is replaced in the condition by $r_{sr,j} \in R_{sr}$, where j is such that $r_{mr,i} \in p_j$. For example, if $p = (p_1, p_2, p_3, p_4) = (\{r_{mr,1}, r_{mr,3}\}, \{r_{mr,2}\}, \{r_{mr,4}\}, \emptyset)$ and $\phi_{mr} = \phi(r_{mr,4})$, then, $\phi_{sr} = \phi(r_{sr,3})$, since $r_{mr,4} \in p_3$. Notice that only one such j exists because p is a partition and, by definition, the different sets of a partition do not have common elements.
  - For every $\delta_{mr} \in \Delta_{mr} : \delta_{mr} = q_{mr}, \phi_{mr} \downarrow (\cdots, r_{mr,i}, \cdots) \rightarrow q'_{mr}$ i.e., for every non-ε transition of A_{mr} that also writes to registers, we do the following. For every $p \in P_{R_{mr}}$ we add a transition $\delta_{sr} = (q_{mr}, p), \phi_{sr} \downarrow r_{sr,k} \rightarrow (q'_{mr}, p')$. ϕ_{sr} is

defined as in the previous case. Now, let $R_w = (\cdots, r_{mr,i}, \cdots)$ denote all the write registers. k in $r_{rs,k}$ is defined as the minimal integer such that $p_k \subseteq R_w$. Additionally, p' is defined as follows. $p'_k = p_k \cup R_w$ and $p'_{k'} = p_{k'} \setminus R_w$ for $k' \neq k$. For example, if $p = (p_1, p_2, p_3, p_4) = (\{r_{mr,1}, r_{mr,3}\}, \{r_{mr,2}\}, \{r_{mr,4}\}, \emptyset)$ and $R_w = \{r_{mr,1}, r_{mr,2}, r_{mr,3}\}$ then $k = 1$, since $p_1 = \{r_{mr,1}, r_{mr,3}\} \subset R_w$. We also have that $p' = (p'_1, p'_2, p'_3, p'_4) = (\{r_{mr,1}, r_{mr,2}, r_{mr,3}\}, \emptyset, \{r_{mr,4}\}, \emptyset)$.

We want to show that $\mathscr{L}(A_{mr}) = \mathscr{L}(A_{sr})$. First, assume that $S \in \mathscr{L}(A_{mr})$ for a string S. We will show that $S \in \mathscr{L}(A_{sr})$. Let ρ_{mr} be an accepting run of A_{mr} over S:

$$\begin{aligned}
\rho_{mr} =& [1, q_{mr,1} = q_{mr,s}, v_{mr,1} = \sharp] \overset{\delta_{mr,1}}{\rightarrow} \\
& \cdots \\
& \overset{\delta_{mr,i-1}}{\rightarrow} [i, q_{mr,i}, v_{mr,i}] \overset{\delta_{mr,i}}{\rightarrow} \\
& \cdots \\
& \overset{\delta_{mr,l}}{\rightarrow} [l+1, q_{mr,l+1} \in Q_{mr,f}, v_{mr,l+1}]
\end{aligned}$$

Let ρ_{sr} be a run of A_{sr} over S:

$$\begin{aligned}
\rho_{sr} =& [1, q_{sr,1} = q_{sr,s}, v_{sr,1} = \sharp] \overset{\delta_{sr,1}}{\rightarrow} \\
& \cdots \\
& \overset{\delta_{sr,i-1}}{\rightarrow} [i, q_{sr,i}, v_{sr,i}] \overset{\delta_{sr,i}}{\rightarrow} \\
& \cdots \\
& \overset{\delta_{sr,l}}{\rightarrow} [l+1, q_{sr,l+1}, v_{sr,l+1}]
\end{aligned}$$

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. $p = (p_{i,1}, \cdots, p_{i,w})$ 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 ρ_{sr}, we have reached a state $q_{sr,i}$. The first element of this state must be $q_{mr,i}$, i.e., the state at the i_{th} configuration of ρ_{mr}. The second element must be a partition (of the registers of A_{mr}). The contents of the first register of A_{sr}, $r_{sr,1}$, must be equal to the contents of the registers of the first set in the partition, the contents of the second register $r_{st,2}$ equal to those of the second set, etc. For example, if $p = (p_{i,1}, p_{i,2}, p_{i,3}, p_{i,4}) = (\{r_{mr,1}, r_{mr,3}\}, \{r_{mr,2}\}, \{r_{mr,4}\}, \emptyset)$, then the contents of $r_{sr,1}$ must be the same as those of $r_{mr,1}$ and $r_{mr,3}$. Similarly, the contents of $r_{sr,2}$ and $r_{mr,2}$ must be the same. Similarly for $r_{sr,3}$ and $r_{mr,4}$. We must then show that, if the induction hypothesis holds for the i^{th} configuration, it also holds for the $(i+1)^{th}$ one.

First, assume that $\delta_{mr,i} = q_{mr,i}, \varepsilon \rightarrow q_{mr,i+1}$. Then, from the construction of A_{sr}, we know that there exists $\delta_{sr} \in \Delta_{sr}$ such that $\delta_{sr} = (q_{mr,i}, p), \varepsilon \rightarrow (q_{mr,i+1}, p)$. 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 $\phi_{mr} \neq \varepsilon$. 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 ϕ_{sr} is triggered. We can prove the latter claim about ϕ_{sr} by noticing that ϕ_{mr} is triggered. Now, from the construction, ϕ_{sr} is the same as ϕ_{mr} with its arguments/registers appropriately replaced, as described above. Without loss of generality, assume that ϕ_{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:

$$\phi_{mr} = \phi(r_{mr,1}, \cdots, r_{mr,w})$$

ϕ_{sr} can be written as follows:

$$\phi_{sr} = \phi(r_{sr,i_1}, \cdots, r_{mr,i_w})$$

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 $v_{mr,i}(r_{mr,j}) = v_{sr,i}(r_{sr,i_j})$. Therefore, ϕ_{mr} and ϕ_{sr} essentially have the same arguments. Since ϕ_{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 $\phi_{mr} \neq \varepsilon$ and $R_w \neq \emptyset$. Then, from the construction of A_{sr}, we know that there exists $\delta_{sr} \in \Delta_{sr}$ such that $\delta_{sr} = (q_{mr,i}, p), \phi_{sr} \downarrow r_{rs,k} \rightarrow (q_{mr,i+1}, p')$ and the condition ϕ_{sr} is triggered. The proof for ϕ_{sr} 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 A_{sr} are the same as those of the registers of A_{mr} contained in the j^{th} set in the partition p'. Indeed, this is the case. From the construction, we know that the partition set p_k (reminder: $r_{sr,k}$ is the write register in δ_{sr}) becomes $p'_k = p_k \cup R_w$. Since $p_k \subset R_w$, this means that $p'_k = R_w$. Thus the hypothesis still holds for p'_k, since $r_{sr,k}$ and all registers in R_w will have the same value. The hypothesis also holds for $k' \neq k$. Since $p'_{k'} = p_{k'} \setminus R_w$ and the hypothesis holds for p, this means that $p_{k'}$ had registers with the same contents before the writing to R_w. If we remove from $p_{k'}$ the changed registers to obtain $p'_{k'}$, then $p'_{k'}$ will still have registers with the same contents after the writing to R_w. Additionally, since $r_{sr,k'}$ 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 ρ_{mr} and ρ_{sr}, because $q_{sr,s} = (q_{mr,s}, p_s)$, with $p_s = (R_{mr}, \emptyset, \cdots, \emptyset)$. 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. ρ_{sr} is thus an accepting run of A_{sr} over S.

We have proven that $S \in \mathscr{L}(A_{mr}) \Rightarrow S \in \mathscr{L}(A_{sr})$. The inverse direction, i.e., $S \in \mathscr{L}(A_{sr}) \Rightarrow S \in \mathscr{L}(A_{mr})$, can be proven similarly. We first assume that there is an accepting run ρ_{sr} of A_{sr} over S and then show, in a similar manner, that there exists an accepting runf= ρ_{mr} of A_{mr} over S. ■

B.4 Proof of Theorem 9.2.4

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

Proof. The proof develops along lines similar to the corresponding proof for classical and register automata [88, 128]. 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* $A = (Q, q_s, Q_f, R, \Delta)$, where $Q = \{q_s, q_f\}$, $Q_f = \{q_f\}$, $R = \{r_1\}$, $\Delta = \{\delta\}$ and $\delta = q_s, (\phi_1 \downarrow \{r_1\}) \cdot (\phi_2(r_1)) \rightarrow q_f$. The single transition δ can read two characters at the same time, apply ϕ_1 on the first one, store this character in r_1 and then apply ϕ_2 on the second character and the contents of r_1. 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 B.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* A_g accepts a string S if $S = S_1 \cdot S_2 \cdot \dots \cdot S_k$ and there exists a run $\rho = [1, q_1, v_1] \rightarrow \cdots \rightarrow [i, q_i, v_i] \rightarrow \cdots \rightarrow [l+1, q_{k+1}, v_{k+1}]$, where the state of the first configuration is the start state of A_g ($q_1 = A_g.q_s$), the state of the last configuration is its final state ($q_{k+1} = A_g.q_f$) and for each i there exists a transition δ of A_g such that $\delta = q_i, e_i \rightarrow q_{i+1}$ and $S_i \in \mathscr{L}(e_i, v_i, v_{i+1})$.

We first convert the initial *SRA* A to a *gSRA* A_g as follows. We add a new start state and connect it to the old start state with an ε 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 *gSRA* A_g which will be equivalent in terms of its language to the original *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 *gSRA* will have one start and one final state, connected with a single transition. The *SREM* on this transition is finally returned as the *SREM* corresponding to the initial *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_{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_{rip} was present. We thus have to modify the *SREM* on the transition from q_i to q_j. The modification is the following. Assume, that, in the old automaton, before the removal, the following hold:

- $q_i, e_1 \rightarrow q_{rip}$.
- $q_{rip}, e_2 \rightarrow q_{rip}$.
- $q_i, e_3 \rightarrow q_j$.
- $q_i, e_4 \rightarrow q_j$.

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_{rip}, the *SREM* on the transition from q_i to q_j becomes $(e_1 \cdot (e_2)^* \cdot e_3) + e_4$. See Algorithm 6.

Algorithm 6: 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 $A$.
1  if $|A.Q| = 2$ then
2      return $A$;
3  else
4      Pick an element $q_{rip}$ from $A.Q$ other than $A.q_s$ or $A.q_f$;
5      $Q' \leftarrow Q - \{q_{rip}\}$;
6      $\Delta' \leftarrow \emptyset$;
       /* Assume $\delta(q_i,q_j)$ returns the SREM on the transition from $q_i$ to $q_j$.   */
7      foreach $q_i \in Q' - \{A.q_f\}$ and $q_j \in Q' - \{q_f\}$ do
8          $\delta' \leftarrow q_i, ((e_1 \cdot (e_2)^* \cdot e_3) + e_4) \rightarrow q_j$ for $e_1 = \delta(q_i, q_{rip})$, $e_2 = \delta(q_{rip}, q_{rip})$,
           $e_3 = \delta(q_{rip}, q_j)$, $e_4 = \delta(q_i, q_j)$;
9          $\Delta' \leftarrow \Delta' \cup \{\delta'\}$;
10     $A' \leftarrow (Q', A.q_s, A.q_f, A.R, \Delta')$;
11     return CONVERT($A'$);
```

We now need to prove that A_g with n states ($A_{g,n}$) and A_g with $n-1$ states ($A_{g,n-1}$), as constructed from $A_{g,n}$ via Algorithm 6, are equivalent, i.e., that $S \in \mathscr{L}(A_{g,n}, v, v') \Leftrightarrow S \in \mathscr{L}(A_{g,n-1}, v, v')$. If we can prove this, then the last step of the recursion of Algorithm 6 (Line 2) will give us an automaton that is equivalent to our initial *SRA A*. Moreover, the *SREM* on the single transition of this *gSRA* is obviously the desired *SREM*.

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

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

Now, assume that $q_i \neq q_{rip}$ for all q_i of ρ, 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 ρ, there exists a transition that is triggered between q_i and q_j. Let e_4 denote the *SREM* on this transition. In $A_{g,n-1}$ the *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 of the term e_4, assuming the same valuation v_i. We can inductively prove, based on the length of the run, that this holds for any two successive configurations. It obviously holds for the first and second configurations, since we start with empty registers and thus the same valuation. As a result, v_2 would also be the same. Inductively, we can prove the same for v_3, etc.

The other case is when $q_i = q_{rip}$ for some i (or multiple is) in ρ. We would then have in ρ a sequence of successive configurations like the following:

$$[i, q_i, v_i] \rightarrow [j, q_{rip}, v_j] \rightarrow \cdots \rightarrow [\cdots, q_{rip}, \cdots] \rightarrow \cdots \rightarrow [k, q_{rip}, v_k] \rightarrow [l, q_l, v_l]$$

We claim that, if we remove from ρ 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 ρ 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 ρ. If there are multiple such sequences in ρ, we can repeat the same process as many times as necessary.

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

Conversely, assume that $S \in \mathscr{L}(A_{g,n-1}, v, v')$. There is then an accepting run ρ 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, $S \in \mathscr{L}(A_{g,n-1}, v, v') \Rightarrow S \in \mathscr{L}(A_{g,n}, v, v')$. This completes our proof.

■

B.5 Proof of Theorem 9.2.5

Theorem *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 B.1d. For union, we construct the *SRA* as in Figure B.1e. For Kleene-star, we construct the *SRA* as in Figure B.1f. 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 $\mathscr{L}(A) = \mathscr{L}(A_1) \cdot \mathscr{L}(A_2)$ then we connect A_1's final state to A_2's start state via an ϵ transition. It is easy to see that if $S_1 \in \mathscr{L}(A_1)$ and $S_2 \in \mathscr{L}(A_2)$ then $S = S_1 \cdot S_2 \in \mathscr{L}(A)$. 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 wan to construct a *SRA* $A = (Q, q_s, Q_f, R, \Delta)$ such that $\mathscr{L}(A) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$. 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 $R_1 \cap R_2 = \emptyset$.
- For each $q = (q_1, q_2) \in Q$ we add a transition δ to $q' = (q'_1, q'_2) \in Q$ if there exists a transition δ_1 from q_1 to q'_1 in A_1 and a transition δ_2 from q_2 to q'_2 in A_2. The condition of δ is $\phi = \delta_1.\phi \wedge \delta_2.\phi$. The write registers of δ are $W = \delta_1.W \cup \delta_2.W$ (notice that, if $\delta_1.W \neq \emptyset$ and $\delta_2.W \neq \emptyset$, this creates a multi-register *SRA*, even if A_1 and A_2 are single-register). Thus, $\delta = (q_1, q_2), (\delta_1.\phi \wedge \delta_2.\phi) \downarrow (\delta_1.W \cup \delta_2.W) \rightarrow (q'_1, q'_2)$.

It is evident that, if a 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, $\mathscr{L}(A) = \mathscr{L}(A_1) \cap \mathscr{L}(A_2)$.

Since *SRA* and *SREM* are equivalent, *SREM* are also closed under union, intersection, concatenation and Kleene-star. ■

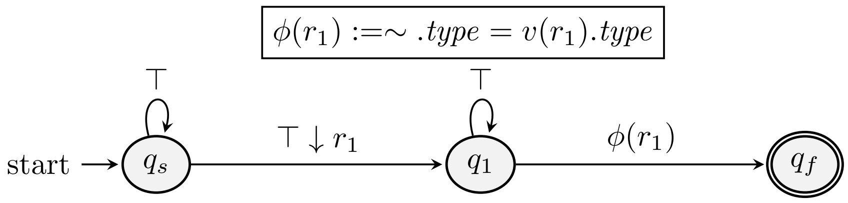


Figure B.2: *SRA* accepting strings which have the same type in two elements. Notice that $\sim$ denotes the current event (last event read from the string).

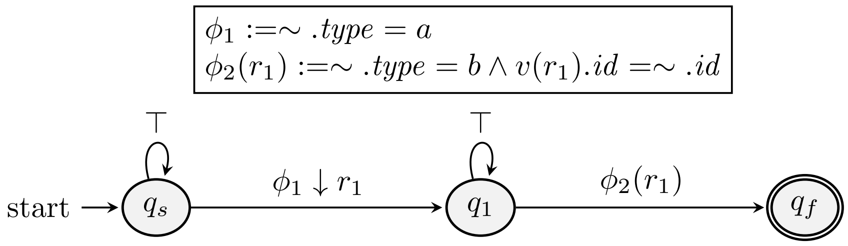

$\phi_1 :=\sim .type = a$
$\phi_2(r_1) :=\sim .type = b \wedge v(r_1).id =\sim .id$
$\top$
$\top$
start
q_s
$\phi_1 \downarrow r_1$
q_1
$\phi_2(r_1)$
q_f

Figure B.3: *SRA* accepting all strings containing an a element followed by a b element, whose identifiers are the same.

B.6 Proof of Theorem 9.2.6

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

Proof. The proof is by a counter example. Let A denote the *SRA* of Figure B.2. 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. ■

B.7 Proof of Theorem 9.2.7

Theorem *SRA* are not closed under determinization.

Proof. The proof is again by a counter example. Let A denote the *SRA* of Figure B.3. This *SRA* reads strings composed of tuples. Each tuple contains an attribute, called *type*, taking values from a finite or infinite alphabet. It also contains another tuple, called *id*, taking integer values. 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 $S'_3 = (b,2)$ and $S' = (a,2)(a,3)(b,2)$ would not be accepted. Now, let

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

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

B.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 B.8.1 — Walk over *SRA*. A walk w over a *SRA* A is a sequence of transitions $w = <\delta_1, \cdots, \delta_k>$, such that:

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

We say that such a walk is of length k. 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., $W_{A,q} = \{w : w \in W_A \wedge \delta_k.target = q\}$.

Definition B.8.2 — Trail over *SRA*. A trail t over a *SRA* A is a walk $w = <\delta_1, \cdots, \delta_k>$ over A, such that:

- $\forall \delta_i, \delta_j \; \delta_i.source \neq \delta_j.source$
- $\forall \delta_i \; \delta_i.source \neq \delta_i.target$

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., $T_{A,q} = \{t : t \in T_A \wedge \delta_k.target = q\}$.

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

Proposition B.8.1 — 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 (δ'_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

$$w = q_1 \overset{\delta_1}{\to} q_2 \cdots q_i \overset{\delta_i}{\to} q_{i+1} \cdots q_j \overset{\delta_j}{\to} q_{j+1} \cdots q_k \overset{\delta_k}{\to} q_{k+1} \overset{\delta_{k+1}}{\to} q_{k+2}$$

  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 δ_l, $i \leq l < j$. Then we get

$$w' = q_1 \overset{\delta_1}{\to} q_2 \cdots q_i \overset{\delta_j}{\to} q_{j+1} \cdots q_k \overset{\delta_k}{\to} q_{k+1} \overset{\delta_{k+1}}{\to} q_{k+2}$$

    or, equivalently, since $q_i = q_j$

$$w' = q_1 \overset{\delta_1}{\to} q_2 \cdots q_j \overset{\delta_j}{\to} q_{j+1} \cdots q_k \overset{\delta_k}{\to} q_{k+1} \overset{\delta_{k+1}}{\to} q_{k+2}$$

    Notice that w' is indeed a walk, since all its transitions are valid, including the one that stitches together the two sub-walks ($q_j \overset{\delta_j}{\to} q_{j+1}$). 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 B.8.3 — Register appearance in a trail. We say that a register r appears in a trail if there exists at least one transition δ in the trail such that $r \in \delta.W$.

In other words, a trail must write to r to say that r appears in it.

Notice that a run of A over a stream S induces a walk over A.

Definition B.8.4 — Walk induced by a run. If

$$\rho = [1, q_s, v_1] \overset{\delta_1}{\to} [2, q_2, v_2] \overset{\delta_2}{\to} \cdots \overset{\delta_{n-1}}{\to} [n, q_n, v_n]$$

is a run of a *SRA* A, then

$$w_\rho =< \delta_1, \cdots, \delta_{n-1} >$$

is the walk induced by ρ.

B.9 Proof of Lemma 9.2.8

Theorem 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 7 shows how we can construct A_{e_w}. The basic idea is that we first construct as usual the *SRA* A_e for the sub-expression e (and eliminate ϵ-transitions). We can then use A_e to enumerate all the possible walks of A_e of length up to w and then join them in a single *SRA* through disjunction. 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 8. 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.

Algorithm 7: Constructing *SRA* for a windowed *SREM* (*ConstructWSRA*).

```
Input: Windowed SREM $e' := e^{[1..w]}$
Output: SRA $A_{e'}$ equivalent to $e'$
1 $A_{e,\epsilon} \leftarrow ConstructSRA(e)$; // As described in Appendix B.1.
2 $A_{e,ms} \leftarrow EliminateEpsilon(A_{e,\epsilon})$; // See Algorithm 5. $A_{e,ms}$ might be multi-register.
3 $A_e \leftarrow ConvertToSingleRegister(A_{e,ms})$; // As described in Appendix B.3.
4 $A_{e'} \leftarrow Unroll(A_e, w)$; // See Algorithm 8.
5 return $A_{e'}$;
```

Algorithm 8: 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     $(A_k, Frontier, CopyOfQ, CopyOfR) \leftarrow Unroll0(A)$; // Algorithm 9
3 else
4     $(A_k, Frontier, CopyOfQ, CopyOfR) \leftarrow UnrollK(A, k)$; // Algorithm 10
5 end
6 return $(A_k, Frontier, CopyOfQ, CopyOfR)$;
```

Algorithm 9: Unrolling cycles for windowed *SREM*, base case (*Unroll0*).

```
    Input: SRA A
    Output: SRA $A_0$ with runs of length 0
1  $q \leftarrow CreateNewState()$;
2  $CopyOfQ \leftarrow \{q \rightarrow A.q_s\}$;
3  $CopyOfR \leftarrow \emptyset$;
4  $Frontier \leftarrow \{q\}$;
5  $Q_f \leftarrow \emptyset$;
6  if $A.q_s \in A.Q_f$ then
7      $Q_f \leftarrow Q_f \cup \{q\}$;
8  end
9  $A_0 \leftarrow (\{q\}, q, Q_f, \emptyset, \emptyset)$;
10 return $(A_0, Frontier, CopyOfQ, CopyOfR)$;
```

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

We first prove the following proposition:

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

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

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

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

such that:

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

We say that a register r appears in a run at position i if $r \in \delta_i.W$, i.e., if the i^{th} transition writes to r. We say that a register r_{e_w}, where $CopyOfR(r_{e_w}) = r_e$, appears last if no other copies of r_e appear after r_{e_w} in a run. The notion of a register's (last) appearance also applies for walks of A_{e_w}, since A_{e_w} is a directed acyclic graph, as can be seen by Algorithms 9 and 10 (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 ρ_e exists). The other is similar.

Base case: $k = 0$. For both *SRA*, only the start state and the initial configuration with all registers empty is possible. Thus, $v_{e,i} = v_{e_w,i} = \sharp$ for all registers. By Algorithm 9 (line 2), we know that $CopyOf(q_{e_w,s}) = q_{e,s}$.

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

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

Algorithm 10: 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 $k$
1  $(A_{k-1}, Frontier, CopyOfQ, CopyOfR) \leftarrow Unroll(A, k-1)$;
2  $NextFrontier \leftarrow \emptyset$;
3  $Q_k \leftarrow A_{k-1}.Q$; $Q_{k,f} \leftarrow A_{k-1}.Q_f$; $R_k \leftarrow A_{k-1}.R$; $\Delta_k \leftarrow A_{k-1}.\Delta$;
4  foreach $q \in Frontier$ do
5      $q_c \leftarrow CopyOfQ(q)$;
6      foreach $\delta \in A.\Delta : \delta.source = q_c$ do
7          $q_{new} \leftarrow CreateNewState()$;
8          $Q_k \leftarrow Q_k \cup \{q_{new}\}$;
9          $CopyOfQ \leftarrow CopyOfQ \cup \{q_{new} \rightarrow \delta.target\}$;
10         if $\delta.target \in A.Q_f$ then
11             $Q_{k,f} \leftarrow Q_{k,f} \cup \{q_{new}\}$;
12         if $\delta.W = \emptyset$ then
13             $R_{new} \leftarrow \emptyset$;
14         else
15             $r_{new} \leftarrow CreateNewRegister()$;
16             $R_k \leftarrow R_k \cup \{r_{new}\}$;
17             $R_{new} \leftarrow \{r_{new}\}$;
18             $CopyOfR \leftarrow CopyOfR \cup \{r_{new} \rightarrow \delta.r\}$; // $\delta.r$ single element of $\delta.W$
19         $\phi_{new} \leftarrow \delta.\phi$;
20         $rs_{new} \leftarrow ()$;
           /* By $\delta.\phi.rs$ we denote the register selection of $\delta.\phi$, i.e., all the
              registers referenced by $\delta.\phi$ in its arguments. $rs$ is represented as a
              list.                                                                        */
21         foreach $r \in \delta.\phi.rs$ do
               /* $FindLastAppearance$ returns a register that is a copy of $r$ and appears
                  last in the trail of $A_{k-1}$ to $q$ (no other copies of $r$ appear after
                  $r_{latest}$). Due to the construction, only a single walk/trail to $q$
                  exists.                                                                  */
22             $r_{latest} \leftarrow FindLastAppearance(r, q, A_{k-1})$;
               /* $::$ denotes the operation of appending an element at the end of a list.
                  $r_{latest}$ is appended at the end of $rs_{new}$.                       */
23             $rs_{new} \leftarrow rs_{new} :: r_{latest}$;
24         $\delta_{new} \leftarrow CreateNewTransition(q, \phi_{new}(rs_{new}) \downarrow R_{new} \rightarrow q_{new})$;
25         $\Delta_k \leftarrow \Delta_k \cup \{\delta_{new}\}$;
26         $NextFrontier \leftarrow NextFrontier \cup \{q_{new}\}$;
27 $A_k \leftarrow (Q_k, A_{k-1}.q_s, Q_{k,f}, R_k, \Delta_k)$;
28 return $(A_k, NextFrontier, CopyOfQ, CopyOfR)$;
```

and

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

be the runs of A_e and A_{e_w} respectively of length $k+1$ over the same $k+1$ elements of a string S. We know that $\rho_{e,k+1}$ is an actual run and we need to construct $\rho_{e_w,k+1}$, knowing, by the induction hypothesis, that there is an actual run up to $q_{e_w,i+k}$. Now, by the construction algorithm, we can see that if $\delta_{e,k}$ is a transition of A_e from $q_{e,k}$ to $q_{e,k+1}$, there exists a transition $\delta_{e_w,k}$ with the same condition from $q_{e_w,k}$ to a $q_{e_w,k+1}$ such that $CopyOfQ(q_{e_w,k+1}) = q_{e,k+1}$. Moreover, if $\delta_{e,k}$ is triggered, so does $\delta_{e_w,k}$, because the registers in the register selection of $\delta_{e_w,k}$ are copies of the corresponding registers in $\delta_{e,k}.\phi.rs$. By the induction hypothesis, we know that the contents of the registers in $\delta_{e,k}.\phi.rs$ will be equal to the contents of their corresponding registers in ρ_{e_w} that appear last. But these are exactly the registers in $\delta_{e_w,k}.\phi.rs$ (see line 22 in Algorithm 10). We can also see that the part of the proposition concerning the valuations v also holds. If $\delta_{e,k}.W = \{r_e\}$ and $\delta_{e_w,k}.W = \{r_{e_w}\}$, then we know, by the construction algorithm (line 18), that $CopyOfR(r_{e_w}) = r_e$ and r_{e_w} will be the last appearance of a copy of r_e in $\rho_{e_w,k+1}$. Thus the proposition holds for $0 < k+1 \leq w$ as well. ■

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

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

B.10 Proof of Theorem 9.2.9

Theorem 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 11. 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 B.7, where we showed that *SRA* are not in general determinizable.

First, we will prove the following proposition:

Proposition B.10.1 There exists a run ρ_n over a string S which A_n can follow by reading the first k tuples of S, iff there exists a run ρ_d that A_d can follow by reading the same first k tuples, such that, if

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

and

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

are the runs of A_n and A_d respectively, then,

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

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

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

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

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

$$\rho_{n,k+1} = \cdots [k, q_{n,k}, v_{n,k}] \begin{cases} \overset{\delta^1_{n,k}}{\rightarrow} [k+1, q^1_{n,k+1}, v^1_{n,k+1}] \\ \overset{\delta^2_{n,k}}{\rightarrow} [k+1, q^2_{n,k+1}, v^2_{n,k+1}] \\ \cdots \\ \overset{\delta^m_{n,k}}{\rightarrow} [k+1, q^m_{n,k+1}, v^m_{n,k+1}] \end{cases} \tag{B.4}$$

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

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

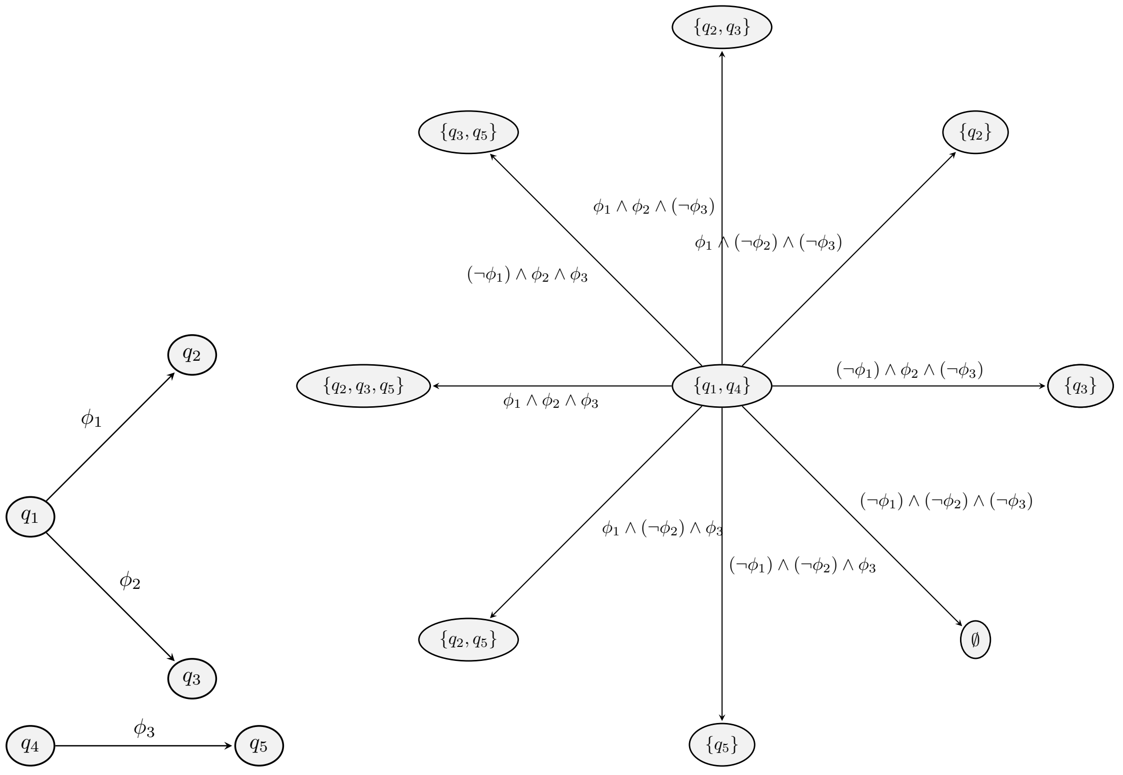
$$\rho_{d,k+1} = \cdots [k, q_{d,k}, v_{d,k}] \overset{\delta_{d,k}}{\rightarrow} [k+1, q_{d,k+1}, v_{d,k+1}]$$

By the induction hypothesis, we know that $q_{n,k} \in q_{d,k}$. By the construction Algorithm 11, we then know that, if $\phi^j_{n,k} = \delta^j_{n,k}.\phi$ is the condition of a transition that takes the non-deterministic run to $q^j_{n,k+1}$, then there exists a transition $\delta_{d,k}$ in the *dSRA* from $q_{d,k}$ whose condition will be a minterm, containing all the $\phi_{n,k}$ in their positive form and all other possible conditions in their negated form. Moreover, the target of that transition, $q_{d,k+1}$, contains all $q^j_{n,k+1}$. More formally, $q_{d,k+1} = \bigcup_{j=1}^{m} q^j_{n,k+1}$.

As an example, see Figure B.4. Figure B.4a depicts part of a *nSRA*. Figure B.4b depicts part of the *dSRA* that woyld be constructed from that of Figure B.4a. The construction

Algorithm 11: Determinization.

```
Input: Windowed SREM $e' := e^{[1..n]}$
Output: Deterministic SRA $A_d$ equivalent to $e'$
1  $A_n \leftarrow ConstructWSRA(e')$; // See Algorithm 7
2  $Q_d \leftarrow ConstructPowerSet(A_n.Q)$;
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      $Conditions \leftarrow ()$; $rs_d \leftarrow ()$;
8      foreach $q_n \in q_d$ do
9          foreach $\delta_n \in A_n.\Delta : \delta_n.source = q_n$ do
10             $Conditions \leftarrow Conditions :: \delta_n.\phi$;
11             $rs_d \leftarrow rs_d :: \delta_n.\phi.rs$;
       /* ConstructMinTerms returns the min-terms from a set of conditions.
          For example, if $Conditions = (\phi_1, \phi_2)$, then
          $MinTerms = (\phi_1 \wedge \phi_2, \neg\phi_1 \wedge \phi_2, \phi_1 \wedge \neg\phi_2, \neg\phi_1 \wedge \neg\phi_2)$   */
12     $MinTerms \leftarrow ConstructMinTerms(Conditions)$;
13     foreach $mt \in MinTerms$ do
14         $p_d \leftarrow \emptyset$; $W_d \leftarrow \emptyset$;
15         foreach $q_n \in q_d$ do
16             foreach $\delta_n \in A_n.\Delta : \delta_n.source = q_n$ do
                   /* $\phi \vDash \psi$ denotes entailment, i.e., if $\phi$ is true then $\psi$ is
                      necessarily also true.  For example, $\phi_1 \wedge \neg\phi_2 \vDash \phi_1$.   */
17                 if $mt \vDash \delta_n.\phi$ then
18                     $p_d \leftarrow p_d \cup \{\delta_n.target\}$;
19                     $W_d \leftarrow W_d \cup \{\delta_n.W\}$;
20         $\delta_d \leftarrow CreateNewTransition(q_d, mt(rs_d) \downarrow W_d \rightarrow p_d)$;
21         $\Delta_d \leftarrow \Delta_d \cup \{\delta_d\}$;
22 $q_{d,s} \leftarrow \{A_n.q_s\}$;
23 $A_d \leftarrow (Q_d, q_{d,s}, Q_{f,d}, A_N.R, \Delta_d)$;
24 return $A_d$;
```


q_1
q_2
q_3
q_4
q_5
ϕ_1
ϕ_2
ϕ_3
$\{q_2, q_3\}$
$\{q_3, q_5\}$
$\{q_2\}$
$\phi_1 \wedge \phi_2 \wedge (\neg\phi_3)$
$\phi_1 \wedge (\neg\phi_2) \wedge (\neg\phi_3)$
$(\neg\phi_1) \wedge \phi_2 \wedge \phi_3$
$\{q_2, q_3, q_5\}$
$\{q_1, q_4\}$
$(\neg\phi_1) \wedge \phi_2 \wedge (\neg\phi_3)$
$\{q_3\}$
$\phi_1 \wedge \phi_2 \wedge \phi_3$
$(\neg\phi_1) \wedge (\neg\phi_2) \wedge (\neg\phi_3)$
$\phi_1 \wedge (\neg\phi_2) \wedge \phi_3$
$(\neg\phi_1) \wedge (\neg\phi_2) \wedge \phi_3$
$\{q_2, q_5\}$
$\emptyset$
$\{q_5\}$

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

Figure B.4: Example of converting a *nSRA* to a *dSRA*.

algorithm would create the state $\{q_2, q_4\}$, the minterms from the conditions of all the outgoing transitions of q_2 and q_4 and then attempt to determine which minterm would move the *dSRA* to which subset of $\{q_2, q_3, q_5\}$. The results is shown in Figure B.4b. 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. (B.4) for the first of these runs and $q_{n,k} = q_4$ for the second. Assume also that both ϕ_1 and ϕ_2 are triggered after reading the $(k+1)^{th}$ element, but not ϕ_3. This means that the *nSRA* would move to q_2 and q_3. In Eq. (B.4), this would mean that $m = 2$ and that $\delta^1_{n,k}.\phi = \phi_1$ and $\delta^2_{n,k}.\phi = \phi_2$. But in the *dSRA* there is a transition that simulates this move of the *nSRA*. The minterm $\phi_1 \wedge \phi_2 \wedge (\neg\phi_3)$ moves the *dSRA* to $\{q_2, q_3\}$. It contains $\delta^1_{n,k}.\phi$ and $\delta^2_{n,k}.\phi$ in their positive form and all other conditions (here only ϕ_3) in their negated form. With a similar reasoning, we see that the *dSRA* can simulate the *nSRA* for every other possible combination of $\{\phi_1, \phi_2, \phi_3\}$.

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

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

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

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

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

The third part also holds. This is the part that actually ensures that the contents of the registers are the same. First, note that a register can appear only once in a run of A_n, because of its tree-like structure. Second, by the construction, we know that $\delta_{d,k}.W = \bigcup_{j=1}^{m} \delta^j_{n,k}.W$ (see again line 19 in the algorithm). Therefore, we know that $\delta_{d,k}$ will write only to registers that had not appeared before in the run of the *nSRA* and will leave every other register that had appeared unaffected. This observation is critical. We could not claim the same for non-windowed *SRA*, as in Figure B.3. If we attempted to determinize this *nSRA*, without unrolling its cycles, the resulting *SRA* could overwrite r_1. Now, since $\delta_{d,k}$ and all the $\delta^j_{n,k}$ write the same element and $\delta_{d,k}$ does not affect any previously appearing registers, the proposition holds. ■

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

B.11 Proof of Corollary 9.2.10

Corollary 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_{dead} (non-final) to A_d. See Algorithm 12. For each state q of A_d, we then gather all the conditions on its outgoing transitions. Let Φ denote this set of conditions. We can then create the conjunction of all the negated conditions in Φ: $\phi_{dead} := (\neg\phi_1) \wedge (\neg\phi_2) \wedge \cdots \wedge (\neg\phi_n)$, where $\phi_i \in \Phi$ and $\bigcup_{i=1}^{n} \phi_i = \Phi$. We then add a transition from q to q_{dead} with ϕ_{dead} as its condition and $\emptyset$ as its

Algorithm 12: Constructing the complement of a *SRA* (*Complement*).

```
Input: Windowed SRA $A$
Output: SRA $A_{complement}$ accepting the complement of $A$'s language
1  $A_d \leftarrow Determinize(A)$; // See Algorithm 11.
2  $q_{dead} \leftarrow CreateNewState()$;
3  $\Delta_{dead} \leftarrow \emptyset$;
4  foreach $q \in A_d.Q$ do
5      $\Phi \leftarrow \emptyset$;
6      foreach $\delta \in A_d.\Delta : \delta.source = q$ do
7          $\Phi \leftarrow \Phi \cup \delta.\phi$;
8      end
9      $\phi_{dead} \leftarrow \top$;
10     foreach $\phi_i \in \Phi$ do
11         $\phi_{dead} \leftarrow \phi_{dead} \wedge (\neg\phi_i)$;
12     end
13     $\delta_{dead} \leftarrow CreateNewTransition(q, \phi_{dead} \downarrow \emptyset \rightarrow q_{dead})$;
14     $\Delta_{dead} \leftarrow \Delta_{dead} \cup \delta_{dead}$;
15 end
16 $\delta_{loop} \leftarrow CreateNewTransition(q_{dead}, \top \downarrow \emptyset \rightarrow q_{dead})$;
17 $\Delta_{dead} \leftarrow \Delta_{dead} \cup \delta_{loop}$;
18 $Q_{comp} \leftarrow A.Q \cup \{q_{dead}\}$;
19 $q_{comp,s} \leftarrow A.q_s$;
20 $Q_{comp,f} \leftarrow A.Q \setminus A.Q_f$;
21 $R_{comp} \leftarrow A.R$;
22 $\Delta_{comp} \leftarrow A.\Delta \cup \Delta_{dead}$;
23 $A_{complement} \leftarrow (Q_{comp}, q_{comp,s}, Q_{comp,f}, R_{comp}, \Delta_{comp})$;
24 return $A_{complement}$;
```

write registers. If we do this for every state $q \in A_d.Q$, we will have created a *SRA* that is equivalent to A_d, since transitions to q_{dead} are only triggered if none of the other conditions in Φ are triggered. If there exists a condition ϕ_i that is triggered, the new automaton will behave exactly as A_d and if no ϕ is triggered it will go to q_{dead}. Now, if we add a self-loop transition on q_{dead} with $\top$ as its condition, we also ensure that the new automaton will always stay in q_{dead} once it enters it. q_{dead} thus acts as a sink state. This new automaton $A_{d,c}$ will therefore be equivalent to A_d and it will also be both deterministic and complete.

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

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

B.12 Proof of Proposition 9.3.1

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

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

C. Bibliography

Conference papers

[3] Jagrati Agrawal et al. “Efficient pattern matching over event streams”. In: *SIGMOD Conference*. ACM, 2008, pages 147–160 (cited on pages 13, 17, 120, 122).

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. “Mining Association Rules between Sets of Items in Large Databases”. In: *SIGMOD Conference*. ACM Press, 1993, pages 207–216 (cited on page 31).

[5] Adnan Akbar et al. “Predicting complex events for pro-active IoT applications”. In: *WF-IoT*. IEEE Computer Society, 2015, pages 327–332 (cited on page 33).

[6] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. “Event Forecasting with Pattern Markov Chains”. In: *DEBS*. ACM, 2017, pages 146–157 (cited on pages 6, 33, 60, 75, 93, 99, 123).

[8] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. “Wayeb: a Tool for Complex Event Forecasting”. In: *LPAR*. Volume 57. EPiC Series in Computing. EasyChair, 2018, pages 26–35 (cited on pages 6, 17, 26, 33, 75, 99, 123).

[21] Alexander Artikis et al. “A Prototype for Credit Card Fraud Management: Industry Paper”. In: *DEBS*. ACM, 2017, pages 249–260 (cited on pages 4, 101).

[22] Alexander Artikis et al. “Complex Event Recognition Languages: Tutorial”. In: *DEBS*. 2017 (cited on page 9).

[23] Howard Barringer et al. “Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors”. In: *FM*. Volume 7436. Lecture Notes in Computer Science. Springer, 2012, pages 68–84 (cited on pages 16, 156).

[25] Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. “Automata with Group Actions”. In: *LICS*. IEEE Computer Society, 2011, pages 355–364 (cited on page 156).

[27] William Brendel, Alan Fern, and Sinisa Todorovic. “Probabilistic event logic for interval-based event recognition”. In: *CVPR*. IEEE Computer Society, 2011, pages 3329–3336 (cited on page 28).

[29] Iliano Cervesato and Angelo Montanari. “A Calculus of Macro-Events: Progress Report”. In: *TIME*. IEEE Computer Society, 2000, pages 47–58 (cited on pages 11, 15).

[31] Sirish Chandrasekaran et al. “TelegraphCQ: Continuous Dataflow Processing”. In: *SIGMOD Conference*. ACM, 2003, page 668 (cited on page 17).

[32] Buru Chang et al. “Content-Aware Hierarchical Point-of-Interest Embedding Model for Successive POI Recommendation”. In: *IJCAI*. ijcai.org, 2018, pages 3301–3307 (cited on pages 31, 123).

[33] Feng Chen and Grigore Rosu. “Parametric Trace Slicing and Monitoring”. In: *TACAS*. Volume 5505. Lecture Notes in Computer Science. Springer, 2009, pages 246–261 (cited on pages 13, 58).

[35] Christine Choppy, Olivier Bertrand, and Patrice Carle. “Coloured Petri Nets for Chronicle Recognition”. In: *Ada-Europe*. Volume 5570. Lecture Notes in Computer Science. Springer, 2009, pages 266–281 (cited on page 21).

[36] Maximilian Christ, Julian Krumeich, and Andreas W. Kempa-Liehr. “Integrating Predictive Analytics into Complex Event Processing by Using Conditional Density Estimations”. In: *EDOC Workshops*. IEEE Computer Society, 2016, pages 1–8 (cited on pages 4, 32, 123).

[40] Gianpaolo Cugola and Alessandro Margara. “TESLA: a formally defined event specification language”. In: *DEBS*. ACM, 2010, pages 50–61 (cited on pages 23, 122).

[44] Loris D’Antoni and Margus Veanes. “The Power of Symbolic Automata and Transducers”. In: *CAV (1)*. Volume 10426. Lecture Notes in Computer Science. Springer, 2017, pages 47–67 (cited on pages 24, 55, 56, 63, 65, 67, 68, 70, 72, 73, 120, 122, 139).

[45] Alan J. Demers et al. “Towards Expressive Publish/Subscribe Systems”. In: *EDBT*. Volume 3896. Lecture Notes in Computer Science. Springer, 2006, pages 627–644 (cited on pages 17, 19).

[46] Alan J. Demers et al. “Cayuga: A General Purpose Event Monitoring System”. In: *CIDR*. www.cidrdb.org, 2007, pages 412–422 (cited on pages 17, 19, 120, 122).

[48] Luping Ding et al. “Runtime Semantic Query Optimization for Event Stream Processing”. In: *ICDE*. IEEE Computer Society, 2008, pages 676–685 (cited on page 26).

[49] Carlotta Domeniconi et al. “A Classification Approach for Prediction of Target Events in Temporal Sequences”. In: *PKDD*. Volume 2431. Lecture Notes in Computer Science. Springer, 2002, pages 125–137 (cited on page 33).

[50] Christophe Dousson. “Extending and Unifying Chronicle Representation with Event Counters”. In: *ECAI*. IOS Press, 2002, pages 257–261 (cited on page 20).

[51] Christophe Dousson, Paul Gaborit, and Malik Ghallab. “Situation Recognition: Representation and Algorithms”. In: *IJCAI*. Morgan Kaufmann, 1993, pages 166–174 (cited on page 20).

[52] Christophe Dousson and Pierre Le Maigat. “Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization”. In: *IJCAI*. 2007, pages 324–329 (cited on pages 20, 34, 122).

[53] Yagil Engel and Opher Etzion. “Towards proactive event-driven computing”. In: *DEBS*. ACM, 2011, pages 125–136 (cited on pages 4, 32, 123).

[57] Lina Fahed, Armelle Brun, and Anne Boyer. “Efficient Discovery of Episode Rules with a Minimal Antecedent and a Distant Consequent”. In: *IC3K (Selected Papers)*. Volume 553. Communications in Computer and Information Science. Springer, 2014, pages 3–18 (cited on page 31).

[58] Chiara Di Francescomarino et al. “Predictive Process Monitoring Methods: Which One Suits Me Best?” In: *BPM*. Volume 11080. Lecture Notes in Computer Science. Springer, 2018, pages 462–479 (cited on page 32).

[60] Lajos Jeno Fülöp et al. “Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics”. In: *BCI*. ACM, 2012, pages 26–31 (cited on pages 4, 32, 123).

[61] Antony Galton and Juan Carlos Augusto. “Two Approaches to Event Definition”. In: *DEXA*. Volume 2453. Lecture Notes in Computer Science. Springer, 2002, pages 547–556 (cited on pages 11, 15, 20).

[62] Malik Ghallab. “On Chronicles: Representation, On-line Recognition and Learning”. In: *KR*. Morgan Kaufmann, 1996, pages 597–606 (cited on pages 20, 34, 122).

[66] Alejandro Grez and Cristian Riveros. “Towards Streaming Evaluation of Queries with Correlation in Complex Event Processing”. In: *ICDT*. Volume 155. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 14:1–14:17 (cited on page 143).

[67] Alejandro Grez, Cristian Riveros, and Martín Ugarte. “A Formal Framework for Complex Event Processing”. In: *ICDT*. Volume 127. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 5:1–5:18 (cited on pages 9, 11, 17, 24, 65, 69–71, 120–122, 140, 141, 143).

[68] Alejandro Grez et al. “On the Expressiveness of Languages for Complex Event Recognition”. In: *ICDT*. Volume 155. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 15:1–15:17 (cited on page 122).

[69] Vincenzo Gulisano et al. “The DEBS 2018 Grand Challenge”. In: *DEBS*. ACM, 2018, pages 191–194 (cited on page 58).

[70] Asela Gunawardana, Christopher Meek, and Puyang Xu. “A Model for Temporal Dependencies in Event Streams”. In: *NIPS*. 2011, pages 1962–1970 (cited on page 34).

[72] Daniel Gyllstrom et al. “SASE: Complex Event Processing over Streams (Demo)”. In: *CIDR*. www.cidrdb.org, 2007, pages 407–411 (cited on page 17).

[81] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. “Lazy evaluation methods for detecting complex events”. In: *DEBS*. ACM, 2015, pages 34–45 (cited on pages 21, 27, 53).

[83] Srivatsan Laxman, Vikram Tankasali, and Ryen W. White. “Stream prediction using a generative model based on frequent episodes in event sequences”. In: *KDD*. ACM, 2008, pages 453–461 (cited on pages 4, 31, 123).

[85] Guoli Li and Hans-Arno Jacobsen. “Composite Subscriptions in Content-Based Publish/-Subscribe Systems”. In: *Middleware*. Volume 3790. Lecture Notes in Computer Science. Springer, 2005, pages 249–269 (cited on page 22).

[87] Zhongyang Li, Xiao Ding, and Ting Liu. “Constructing Narrative Event Evolutionary Graph for Script Event Prediction”. In: *IJCAI*. ijcai.org, 2018, pages 4201–4207 (cited on pages 31, 123).

[89] Leonid Libkin and Domagoj Vrgoc. “Regular Expressions for Data Words”. In: *LPAR*. Volume 7180. Lecture Notes in Computer Science. Springer, 2012, pages 274–288 (cited on page 122).

[90] Mo Liu et al. “E-Cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing”. In: *SIGMOD Conference*. ACM, 2011, pages 889–900 (cited on pages 16, 23).

[96] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. "Learning from the past: automated rule generation for complex event processing". In: *DEBS*. ACM, 2014, pages 47–58 (cited on page 28).

[98] Yuan Mei and Samuel Madden. "ZStream: a cost-based query processor for adaptively detecting composite events". In: *SIGMOD Conference*. ACM, 2009, pages 193–206 (cited on pages 22, 122).

[101] Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen. "Predictive publish/subscribe matching". In: *DEBS*. ACM, 2010, pages 14–25 (cited on pages 32, 33, 74, 100, 123).

[105] Emmanouil Ntoulias et al. "Online trajectory analysis with scalable event recognition". In: *EDBT/ICDT Workshops*. Volume 2841. CEUR Workshop Proceedings. CEUR-WS.org, 2021 (cited on page 6).

[110] Suraj Pandey, Surya Nepal, and Shiping Chen. "A test-bed for the evaluation of business process prediction techniques". In: *CollaborateCom*. ICST, 2011, pages 382–391 (cited on pages 33, 100, 123).

[115] Peter R. Pietzuch, Brian Shand, and Jean Bacon. "A Framework for Event Composition in Distributed Systems". In: *Middleware*. Volume 2672. Lecture Notes in Computer Science. Springer, 2003, pages 62–82 (cited on pages 17, 20).

[117] Ehab Qadah et al. "A Distributed Online Learning Approach for Pattern Prediction over Movement Event Streams with Apache Flink". In: *EDBT/ICDT Workshops*. Volume 2083. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pages 109–116 (cited on page 7).

[120] Dana Ron, Yoram Singer, and Naftali Tishby. "The Power of Amnesia". In: *NIPS*. Morgan Kaufmann, 1993, pages 176–183 (cited on pages 30, 63, 75, 76, 120, 146, 147).

[124] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. "Distributed complex event processing with query rewriting". In: *DEBS*. ACM, 2009 (cited on pages 17, 20, 21, 27).

[125] Luc Segoufin. "Automata and Logics for Words and Trees over an Infinite Alphabet". In: *CSL*. Volume 4207. Lecture Notes in Computer Science. Springer, 2006, pages 41–57 (cited on page 122).

[126] Joseph Selman et al. "PEL-CNF: Probabilistic event logic conjunctive normal form for video interpretation". In: *ICCV Workshops*. IEEE Computer Society, 2011, pages 680–687 (cited on page 28).

[132] Margus Veanes. "Applications of Symbolic Finite Automata". In: *CIAA*. Volume 7982. Lecture Notes in Computer Science. Springer, 2013, pages 16–23 (cited on page 122).

[133] Margus Veanes, Nikolaj Bjørner, and Leonardo Mendonça de Moura. "Symbolic Automata Constraint Solving". In: *LPAR (Yogyakarta)*. Volume 6397. Lecture Notes in Computer Science. Springer, 2010, pages 640–654 (cited on pages 122, 139).

[134] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. "Rex: Symbolic Regular Expression Explorer". In: *ICST*. IEEE Computer Society, 2010, pages 498–507 (cited on pages 26, 55, 65).

[135] Ricardo Vilalta and Sheng Ma. "Predicting Rare Events In Temporal Domains". In: *ICDM*. IEEE Computer Society, 2002, pages 474–481 (cited on pages 4, 31, 123).

[136] Marios Vodas et al. "Online Distributed Maritime Event Detection and Forecasting over Big Vessel Tracking Data". In: *IEEE BigData*. IEEE Computer Society, 2021 (cited on page 6).

[137] George A. Vouros et al. “Big Data Analytics for Time Critical Mobility Forecasting: Recent Progress and Research Challenges”. In: *EDBT*. OpenProceedings.org, 2018, pages 612–623 (cited on pages 4, 7).

[138] George A. Vouros et al. “Increasing Maritime Situation Awareness via Trajectory Detection, Enrichment and Recognition of Events”. In: *W2GIS*. Volume 10819. Lecture Notes in Computer Science. Springer, 2018, pages 130–140 (cited on page 6).

[139] Gary M. Weiss and Haym Hirsh. “Learning to Predict Rare Events in Event Sequences”. In: *KDD*. AAAI Press, 1998, pages 359–363 (cited on page 33).

[141] Eugene Wu, Yanlei Diao, and Shariq Rizvi. “High-performance complex event processing over streams”. In: *SIGMOD Conference*. ACM, 2006, pages 407–418 (cited on page 17).

[142] Zhengzheng Xing et al. “Mining Sequence Classifiers for Early Prediction”. In: *SDM*. SIAM, 2008, pages 644–655 (cited on page 34).

[143] Haopeng Zhang, Yanlei Diao, and Neil Immerman. “On complexity and optimization of expensive queries in complex event processing”. In: *SIGMOD Conference*. ACM, 2014, pages 217–228 (cited on pages 11–13, 16–18, 38, 58, 122).

[145] Detlef Zimmer and Rainer Unland. “On the Semantics of Complex Events in Active Database Management Systems”. In: *ICDE*. IEEE Computer Society, 1999, pages 392–399 (cited on pages 12–14).

Journal articles

[1] Wil M. P. van der Aalst, M. H. Schonenberg, and Minseok Song. “Time prediction based on process mining”. In: *Inf. Syst.* 36.2 (2011), pages 450–475 (cited on page 100).

[2] Naoki Abe and Manfred K. Warmuth. “On the Computational Complexity of Approximating Distributions by Probabilistic Automata”. In: *Machine Learning* 9 (1992), pages 205–260 (cited on pages 33, 75, 123).

[7] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. “Symbolic Automata with Memory: a Computational Model for Complex Event Processing”. In: *CoRR* abs/1804.09999 (2018) (cited on page 141).

[9] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. “Complex event forecasting with prediction suffix trees: extended technical report”. In: *CoRR* abs/2109.00287 (2021). URL: `https://arxiv.org/abs/2109.00287` (cited on page 120).

[10] Elias Alevizos, Alexander Artikis, and Georgios Paliouras. “Complex event forecasting with prediction suffix trees”. In: *VLDB J.* 31.1 (2022), pages 157–180 (cited on pages 6, 120, 121, 140, 146, 147).

[11] Elias Alevizos et al. “Probabilistic Complex Event Recognition: A Survey”. In: *ACM Comput. Surv.* 50.5 (2017), 71:1–71:31 (cited on pages 6, 9, 11, 65).

[12] James F. Allen. “Maintaining Knowledge about Temporal Intervals”. In: *Commun. ACM* 26.11 (1983), pages 832–843 (cited on pages 11, 15, 26).

[13] James F. Allen. “Towards a General Theory of Action and Time”. In: *Artif. Intell.* 23.2 (1984), pages 123–154 (cited on pages 11, 15, 26).

[17] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL continuous query language: semantic foundations and query execution”. In: *VLDB J.* 15.2 (2006), pages 121–142 (cited on page 17).

[18] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. “An Event Calculus for Event Recognition”. In: *IEEE Trans. Knowl. Data Eng.* 27.4 (2015), pages 895–908 (cited on pages 15, 16, 21, 122).

[19] Alexander Artikis et al. “Logic-based event recognition”. In: *Knowl. Eng. Rev.* 27.4 (2012), pages 469–506 (cited on page 20).

[20] Alexander Artikis et al. “Scalable Proactive Event-Driven Decision Making”. In: *IEEE Technol. Soc. Mag.* 33.3 (2014), pages 35–41 (cited on page 4).

[24] Ron Begleiter, Ran El-Yaniv, and Golan Yona. “On Prediction Using Variable Order Markov Models”. In: *J. Artif. Intell. Res.* 22 (2004), pages 385–421 (cited on pages 4, 30, 33, 75, 96, 123, 146, 147).

[26] Mikolaj Bojanczyk et al. “Two-variable logic on data words”. In: *ACM Trans. Comput. Log.* 12.4 (2011), 27:1–27:26 (cited on page 122).

[28] Peter Bühlmann, Abraham J Wyner, et al. “Variable length Markov chains”. In: *The Annals of Statistics* 27.2 (1999), pages 480–513 (cited on pages 4, 30, 63, 75).

[34] Chung-Wen Cho et al. “On-line rule matching for event prediction”. In: *VLDB J.* 20.3 (2011), pages 303–334 (cited on pages 4, 31, 123).

[37] John G. Cleary and Ian H. Witten. “Data Compression Using Adaptive Coding and Partial String Matching”. In: *IEEE Trans. Communications* 32.4 (1984), pages 396–402 (cited on pages 4, 30, 75, 123, 146).

[38] Don Coppersmith and Shmuel Winograd. “Matrix Multiplication via Arithmetic Progressions”. In: *J. Symb. Comput.* 9.3 (1990), pages 251–280 (cited on page 164).

[41] Gianpaolo Cugola and Alessandro Margara. “Complex event processing with T-REX”. In: *J. Syst. Softw.* 85.8 (2012), pages 1709–1728 (cited on page 23).

[42] Gianpaolo Cugola and Alessandro Margara. “Processing flows of information: From data stream to complex event processing”. In: *ACM Comput. Surv.* 44.3 (2012), 15:1–15:62. DOI: `10.1145/2187671.2187677` (cited on pages 3, 9, 12, 13, 15, 65, 119, 120, 122).

[43] Loris D’Antoni and Margus Veanes. “Extended symbolic finite automata and transducers”. In: *Formal Methods Syst. Des.* 47.1 (2015), pages 93–119 (cited on pages 24, 65, 156).

[47] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. “Sase+: An agile language for kleene closure over event streams”. In: *UMass Technical Report* (2007) (cited on page 17).

[63] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. “TRIO: A logic language for executable specifications of real-time systems”. In: *J. Syst. Softw.* 12.2 (1990), pages 107–123 (cited on page 23).

[64] Nikos Giatrakos et al. “Complex event recognition in the Big Data era: a survey”. In: *VLDB J.* 29.1 (2020), pages 313–352 (cited on pages 3, 4, 6, 63, 65, 70, 119, 122).

[65] Tilmann Gneiting and Adrian E Raftery. “Strictly proper scoring rules, prediction, and estimation”. In: *Journal of the American Statistical Association* 102.477 (2007), pages 359–378 (cited on page 97).

[73] Sylvain Hallé. “From Complex Event Processing to Simple Event Processing”. In: *CoRR* abs/1702.08051 (2017). URL: `http://arxiv.org/abs/1702.08051` (cited on pages 9, 122).

[79] Michael Kaminski and Nissim Francez. “Finite-Memory Automata”. In: *Theor. Comput. Sci.* 134.2 (1994), pages 329–363 (cited on pages 16, 24, 120, 122, 136, 173).

[80] Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. “Parallel online event calculus learning for complex event recognition”. In: *Future Gener. Comput. Syst.* 94 (2019), pages 468–478 (cited on page 27).

[82] Robert A. Kowalski and Marek J. Sergot. “A Logic-based Calculus of Events”. In: *New Gener. Comput.* 4.1 (1986), pages 67–95 (cited on pages 11, 15, 21).

[84] O-Joun Lee and Jai E. Jung. “Sequence Clustering-based Automated Rule Generation for Adaptive Complex Event Processing”. In: *Future Gener. Comput. Syst.* 66 (2017), pages 100–109 (cited on page 28).

[86] Yan Li, Tingjian Ge, and Cindy X. Chen. “Data Stream Event Prediction Based on Timing Knowledge and State Transitions”. In: *Proc. VLDB Endow.* 13.10 (2020), pages 1779–1792 (cited on pages 33, 123).

[88] Leonid Libkin, Tony Tan, and Domagoj Vrgoc. “Regular expressions for data words”. In: *J. Comput. Syst. Sci.* 81.7 (2015), pages 1278–1297 (cited on pages 120, 122, 123, 128, 142, 176).

[91] Manuel E Lladser, Meredith D Betterton, and Rob Knight. “Multiple pattern matching: A Markov chain approach”. In: *Journal of mathematical biology* 56.1 (2008), pages 51–92 (cited on page 38).

[94] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. “Statistical and Machine Learning forecasting methods: Concerns and ways forward”. In: *PloS one* 13.3 (2018), e0194889 (cited on page 32).

[95] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. “Discovery of Frequent Episodes in Event Sequences”. In: *Data Min. Knowl. Discov.* 1.3 (1997), pages 259–289 (cited on page 31).

[97] Alfonso Eduardo Márquez-Chamorro, Manuel Resinas, and Antonio Ruiz-Cortés. “Predictive Monitoring of Business Processes: A Survey”. In: *IEEE Trans. Services Computing* 11.6 (2018), pages 962–977 (cited on pages 4, 32).

[100] T. Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (Apr. 1989), pages 541–580. ISSN: 0018-9219 (cited on page 26).

[102] Frank Neven, Thomas Schwentick, and Victor Vianu. “Finite state machines for strings over infinite alphabets”. In: *ACM Trans. Comput. Log.* 5.3 (2004), pages 403–435 (cited on page 122).

[103] Pierre Nicodème, Bruno Salvy, and Philippe Flajolet. “Motif statistics”. In: *Theor. Comput. Sci.* 287.2 (2002), pages 593–617 (cited on pages 38, 39).

[104] Gertjan van Noord and Dale Gerdemann. “Finite State Transducers with Predicates and Identities”. In: *Grammars* 4.3 (2001), pages 263–286 (cited on page 122).

[106] Emmanouil Ntoulias et al. “Online fleet monitoring with scalable event recognition and forecasting”. In: *GeoInformatica* (2022) (cited on page 6).

[107] Grégory Nuel. “Pattern Markov chains: optimal Markov chain embedding through deterministic finite automata”. In: *Journal of Applied Probability* 45.1 (2008), pages 226–243 (cited on pages 38, 39, 56).

[109] Jonathan Ozik et al. “Learning-accelerated discovery of immune-tumour interactions”. In: *Molecular systems design & engineering* 4.4 (2019), pages 747–760 (cited on page 4).

[111] Adrian Paschke. “ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics”. In: *CoRR* abs/cs/0610167 (2006) (cited on page 15).

[112] Adrian Paschke and Martin Bichler. “Knowledge representation concepts for automated SLA management”. In: *Decis. Support Syst.* 46.1 (2008), pages 187–205 (cited on page 15).

[113] Kostas Patroumpas et al. “Online event recognition from moving vessel trajectories”. In: *GeoInformatica* 21.2 (2017), pages 389–427 (cited on pages 10, 50, 57, 107).

[118] Lawrence R Rabiner. “A tutorial on hidden Markov models and selected applications in speech recognition”. In: *Proceedings of the IEEE* 77.2 (1989), pages 257–286 (cited on page 75).

[121] Dana Ron, Yoram Singer, and Naftali Tishby. “The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length”. In: *Machine Learning* 25.2-3 (1996), pages 117–149 (cited on pages 4, 30, 63, 75–77, 91–93, 120, 123, 146, 147).

[140] Frans M. J. Willems, Yuri M. Shtarkov, and Tjalling J. Tjalkens. “The context-tree weighting method: basic properties”. In: *IEEE Trans. Information Theory* 41.3 (1995), pages 653–664 (cited on pages 4, 30, 75, 123, 146).

[144] Cheng Zhou, Boris Cule, and Bart Goethals. “A pattern based predictor for event streams”. In: *Expert Syst. Appl.* 42.23 (2015), pages 9294–9306 (cited on pages 4, 31, 123).

Books

[30] Ugur Çetintemel et al. *The Aurora and Borealis Stream Processing Engines*. Data-Centric Systems and Applications. Springer, 2016, pages 337–359 (cited on page 15).

[39] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007 (cited on pages 38, 39).

[56] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Company, 2010. ISBN: 978-1-935182-21-4 (cited on pages 10, 13, 15, 24, 64, 119).

[59] James C Fu and WY Wendy Lou. *Distribution theory of runs and patterns and its applications: a finite Markov chain imbedding approach*. World Scientific, 2003 (cited on pages 38, 41, 84).

[71] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997 (cited on pages 38, 39).

[74] Shawn Hedman. *A First Course in Logic: An introduction to model theory, proof theory, computability, and complexity*. Oxford University Press Oxford, 2004 (cited on pages 124, 125).

[75] Ulrich Hedtstück. *Complex event processing: Verarbeitung von Ereignismustern in Datenströmen*. Berlin: Springer Vieweg, 2017 (cited on pages 13, 64, 119).

[76] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007 (cited on pages 24, 38, 39, 67, 68, 171).

[92] M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, 2005 (cited on pages 38, 42).

[93] David C. Luckham. *The power of events - an introduction to complex event processing in distributed enterprise systems*. ACM, 2005 (cited on pages 10, 64, 119).

[99] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015 (cited on pages 4, 29, 123).

[116] Teodor C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*. Morgan Kaufmann, 1988, pages 193–216 (cited on page 21).

[122] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009. ISBN: 978-0-521-84425-3 (cited on pages 57, 74, 146).

[128] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997 (cited on page 176).

[130] Andrew James Stothers. *On the complexity of matrix multiplication (PhD thesis)*. 2010 (cited on page 164).

[131] Wil Van Der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011 (cited on page 32).

Misc

[14] *Apache Flink v. 1.7*. `https://flink.apache.org/`. [Online; accessed 31-March-2019] (cited on page 13).

[15] *Apache FlinkCEP*. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html`. [Online; accessed 31-March-2019] (cited on pages 13, 17–19).

[16] *Apache Spark Streaming*. `https://spark.apache.org/docs/latest/streaming-programming-guide.html`. [Online; accessed 31-March-2019] (cited on page 19).

[54] *Esper*. `http://www.espertech.com/esper`. [Online; accessed 31-March-2019] (cited on pages 23, 100).

[55] Opher Etzion. *Proactive Computing: Changing the Future*. `https://www.rtinsights.com/proactive-computing-prescriptive-analytics-special-report/`. [Online; accessed 23-February-2017]. 2016 (cited on page 4).

[77] *Jess, the Rule Engine for the Java Platform*. `https://www.jessrules.com/jess/docs/71/`. [Online; accessed 31-March-2019] (cited on pages 22, 25).

[78] Anne-Laure Jousselme et al. *Deliverable D5.1: Maritime use case description*. Project datAcron (cited on page 58).

[108] *Oracle CEP CQL Language Reference*. `https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm`. [Online; accessed 31-March-2019] (cited on page 17).

[114] K Patroumpas et al. *Final dataset of Trajectory Synopses over AIS kinematic messages in Brest area (ver. 0.8) [Data set], 10.5281/zenodo.2563256*. 2018. DOI: `10.5281/zenodo.2563256`. URL: `http://doi.org/10.5281/zenodo.2563256` (cited on page 107).

[119] C Ray et al. *Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, 10.5281/zenodo.1167595*. 2018. DOI: `10.5281/zenodo.1167595`. URL: `https://doi.org/10.5281/zenodo.1167595` (cited on pages 57, 107).

[123] *SASE source code*. `https://github.com/haopeng/sase/`. [Online; accessed 31-March-2019] (cited on page 18).

[127] *Siddhi CEP*. `https://github.com/wso2/siddhi`. [Online; accessed 31-March-2019] (cited on page 19).

[129] *Smile - Statistical Machine Intelligence and Learning Engine*. `http://haifengl.github.io/` (cited on page 101).

Index