ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230399

Answer Set Automata: A Learnable Pattern Specification Framework for Complex Event Recognition

Nikos Katzouris[a;*] **and Georgios Paliouras**[a]

[a]Institute of Informatics, National Center for Scientific Research "Demokritos", Athens, Greece
ORCiD ID: Nikos Katzouris https://orcid.org/0000-0001-8804-470X,
Georgios Paliouras https://orcid.org/0000-0001-9629-2367

Abstract. Complex Event Recognition (CER) systems detect event occurrences in streaming input using predefined event patterns. Techniques that learn event patterns from data are highly desirable in CER. Since such patterns are typically represented by symbolic automata, we propose a family of such automata where the transition-enabling conditions are defined by Answer Set Programming (ASP) rules, and which, thanks to the strong connections of ASP to symbolic learning, are learnable from data. We present such a learning approach in ASP, capable of jointly learning the structure of an automaton, while synthesizing the transition guards from building-block predicates, and a scalable, incremental version thereof that progressively revises models learnt from mini-batches using Monte Carlo Tree Search. We evaluate our approach on three CER datasets and empirically demonstrate its efficacy.

1 Introduction

Complex Event Recognition (CER) systems [20, 3] detect occurrences of *complex events* (CEs) in streaming input, using temporal patterns consisting of *simple events*, e.g. sensor data, or other complex events. CE patterns are typically defined by domain experts in some *event specification language* (ESL) [22] . Despite the diversity of such languages, a minimal event processing operators that every ESL should support [39, 4, 20, 21] includes *sequence* and *iteration (Kleene Closure)*, implying respectively that some particular events should succeed one another temporally, or that an event should occur iteratively in a sequence, and the *filtering* operator, which matches input events that satisfy a set of predefined predicates.

Taken together, these three operators point to a computational model for CER based on symbolic finite automata [11] (SFA), i.e., automata where the transition-enabling conditions are predicates than need to be evaluated against the input, rather than mere symbols. As a result, in most existing CER systems CE pattern definitions are SFA-based [38, 1, 39, 16, 14, 15, 37, 35, 10, 3]. Prominent areas of CER research, then, concern the study of trade-offs between ESLs' expressive power and pattern matching complexity [16, 22], in addition to practical issues, such as scalability and distributed processing.

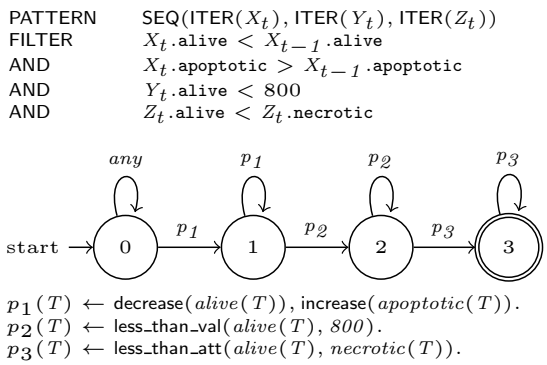
CE pattern learning is a less studied CER topic, which, however, is of utmost importance, since CE patterns are not always known in advance, or they frequently need to be revised. A few learning approaches have been proposed, which have several limitations. Some focus more on learning in the presence of commonsense phenomena, such as the duration of events in time [24, 23], and less on the sequential nature of such events; others do support operators such as iteration [32, 25, 26, 19], or filtering predicates [28, 18], while most offer very limited support for reasoning with background knowledge and CE pattern revision.

To address such issues we propose *answer set automata learning* (ASAL), a framework that allows to specify SFA-based CE patterns in the form of answer set programs (ASP) [29], which, thanks to the strong connections of ASP to symbolic learning, are directly learnable and revisable from data. ASAL allows to synthesize patterns utilizing the core CER operators by jointly learning the structure of an SFA pattern and the definitions of its transition guards, consisting of Boolean combinations of building-block, background knowledge predicates. The core ASAL approach relies on abduction w.r.t. an SFA interpreter. To scale it up to large training sets, we utilize SFA revision in a Monte Carlo Tree Search (MCTS) that continuously revises programs learnt from mini-batches of the data, in an effort to approximate a global optimum. We evaluate both the batch and the incremental, MCTS-based versions of our approach on three CER datasets and compare it to classical automata learning techniques, demonstrating empirically its efficacy.

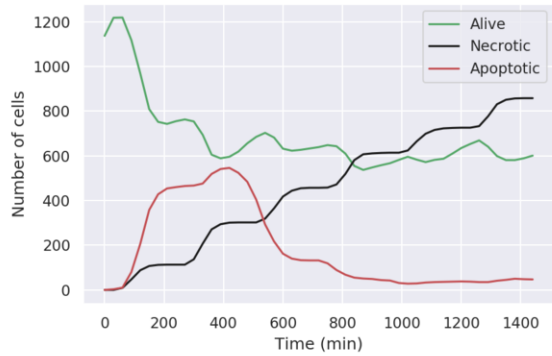
2 Related Work

The methods introduced in [19] and [32] learn event-based patterns from historical traces, along with filtering constraints between the attributes of the constituent events. These methods assume purely sequence-based ESLs that do not support iteration, therefore, they are restricted to simple sequential patterns, rather than SFA-based ones. Closely related is the technique of [25, 26], which extracts CE patterns in the form of frequent queries. However, the number of such queries can be excessive, without them being necessarily representative of the situations (CEs) of interest [26]. Moreover, this technique is also restricted to a purely sequential ESL. The method of [28] precedes the process of constructing a CE pattern in the form of a probabilistic automaton, by a representation learning technique that generates the pattern's constituent events in an unsupervised fashion. This is a a purely learning-based method designed to overcome the unavailability of domain knowledge on informative event primitives. On the downside, this method has no connections to concrete ESLs and learns less interpretable patterns that cannot be used with existing CER engines, since a pattern's constituent events are opaque.

* Corresponding Author. Email: nkatz@iit.demokritos.gr.



(a) A CE pattern that captures the simulation on the right (upper), its corresponding SFA (middle) and the SFA's guards (bottom).



(b) Temporal evolution of different cell populations after the injection of a drug coctail over the course of a simulation [2, 36].

Figure 1: CER for cancer treatment simulation optimization.

ASAL supports the core CER operators of *sequence*, *filtering* and *iteration*, thus going beyond the sequence pattern learning task of [19, 32, 25, 26]. Moreover, in contrast to [28], the programs learnt by ASAL are easily translatable into any ESL that supports the above-mentioned minimum of expressive power.

The field of finite automata (FA) learning [12] has a long history in the literature [5, 34, 27, 6, 18]. Most existing techniques are either noise-intolerant leaners [5, 6], or rely on greedy heuristics for *state merging* [34, 27], a technique that generalizes as much as possible from a large, seed induction structure, the *Prefix Tree Acceptor*, generated from the entire training set. These approaches often tend to learn large, overfitted models that generalize poorly and raise scalability issues in large datasets. In contrast to the above, ASAL learns incrementally from small data batches, never processing the training set in its entirety, while still aiming for a model with an adequate global performance, in a noise-tolerant fashion.

The aforementioned FA induction algorithms learn classical – as opposed to symbolic – FA. Moreover, they all learn from single-sequence input, an important limitation to their applicability in CER, where the input is typically multivariate. On the other hand, although some algorithms for SFA induction do exist [31, 7, 17], they are mostly based on "upgrading" existing classical FA identification techniques to infinite alphabets, and they thus suffer from the limitations outlined above.

Learning FA and grammars has been an application domain for Inductive Logic Programming (ILP) [9, 13] since its early days. More recent ILP frameworks have also been applied to the task [33, 18]. Both these approaches are designed to learn from small univariate training samples and cannot deal with noisy input.

3 Background and Problem Statement

We begin with a brief description of a tumor evolution simulation optimization task [2, 36], which we will refer to as a running example throughout the paper. Figure 1(b) presents the temporal evolution of tumor cell populations of different type (*alive, necrotic, apoptotic*) in a computer simulation, as a result of injecting a tumor necrotic factor (TNF – a drug cocktail) into the tumor. The goal is to assess the efficacy of the particular TNF in limiting tumor growth. Processing such a simulation with a CER system would allow to detect critical events over its course, which in turn may facilitate drug development research. For instance, given that such simulations are extremely demanding computationally, early-stopping unpromising ones, based on the detected events, to devote computational resources elsewhere, can significantly speed-up the research [36].

Event tuples. Typically, CER systems operate on streams of *event tuples* [20, 4], i.e. time-stamped tuples of attribute-value pairs. In general, we can think about CER input as a multivariate sequence with one sub-sequence per event attribute. For instance, an attribute may correspond to a particular sensor and its values to the sensor readings over time, which may be numerical, or categorical. An event tuple, then, is a "snapshot" of the joint evolution of all domain sensor readings over time. As an example, the multivariate sequence simulation input in Figure 1(b) is converted into a sequence of event tuples, with one tuple per time step in the simulation. The tuple corresponding to $t = 200$ would be $\langle necrotic = 110, apoptotic = 420, alive = 770, time = 200\rangle$.

CE patterns define a temporal structure over event tuples and a set of constraints over their attributes. A pattern is matched once a set of event tuples is encountered in the input, such that the tuples' temporal ordering adheres to the pattern's temporal structure and the tuples' attributes satisfy the pattern's constraints.

The upper part of Figure 1(a) presents an example of such a pattern that may be matched against the input of the simulation in Figure 1(b). The pattern is expressed in a pseudo-ESL that illustrates the core CER operators of *sequence, iteration* and *filtering*. The pattern's variables X_t, Y_t, Z_t are assumed to be ranging over event tuples and X_t refers to a tuple received at time t. The first line specifies the temporal structure of the pattern, using the operators SEQ($E_1 \ldots E_n$), which matches any occurrence of tuples $E_1 \ldots E_n$ in a sequence, and ITER(E) (*iteration*), which matches any iterative occurrence of more than one instances [1] of E.

The FILTER/AND part of the pattern defines the constraints that the instances of X_t, Y_t, Z_t should satisfy. The first two lines (following the "PATTERN" part) dictate that for each pair of consecutive tuples X_{t-1} and X_t the value of the *alive* attribute should decrease and that of the *apoptotic* attribute should increase. The next line dictates that any Y_t event tuple instance is expected to respect a threshold on the population size of *alive* cells, while the last line in the pattern dictates that for each Z_t tuple instance, the value of *necrotic* should be lower than that of *alive*. It may then be seen that the entire pattern matches

[1] This is the semantics of the iteration operator that we assume in this work. This is in contrast to other iteration operator semantics, which match *zero or more* occurrences of E.

cases such as those presented in the simulation of Figure 1(b), where (i) initially there is period where the *alive* cancer cells population is constantly decreasing, while that of apoptotic ones is increasing; (ii) this period is followed by another where the *alive* cells population does not exceed a given threshold; (iii) the latter is followed by a last period where the population of *alive* cells is strictly lower than that of *necrotic*. This pattern expresses a common motif of the effects of successful TNFs on tumor growth [2].

From CE patterns to SFA. CE patterns may be converted into SFA by mapping a pattern's temporal structure to the SFA's structure and the pattern's filters to the SFA's transition guards. This is illustrated in the middle and lower parts of Figure 1(a) respectively, where the guards are presented as a set of logic programming rules. Note that the T variable there has the same meaning as the t subscript in the pattern, i.e. to implicitly refer to the tuple received at time T. The SFA loops on its start state until the first occurrence of a p_1-satisfying tuple. The latter is defined as a conjunction of two predicates, decrease/1 and increase/1, which are assumed to be defined as background knowledge (BK) to reflect the simultaneous change in *alive* and *necrotic* cell populations specified by the pattern's filter (we will provide example BK predicate implementations in Section 4). Upon the occurrence of a p_1-satisfying tuple, the SFA moves to state 1, where it loops on additional occurrences of such tuples. The rest of the SFA's functionality is similar.

The learning task that we address in this work is that of jointly inducing the structure of an SFA from labeled sequences of event tuples, and the definitions of the SFA's transition guards from given, BK predicates. By mapping compositions of ITER and SEQ operators to SFA structure, and FILTER constraints to transition guard predicates, learnt SFAs may be translated into ESL specifications and vice-versa, provided that the target ESL supports SEQ and ITER.

Restriction to unary predicates. BK predicates constitute a *language bias* for our learning task. A limitation of our proposed method, which we plan to address in future work, is the fact that such language bias is currently restricted to unary predicates, such as $p_1/1, p_2/1, p_3/1$ in Figure 1(a). Such predicates can only express across-attribute relations within a single event tuple E_t, or across-time relations between the attributes of two different event tuples E_t and E_{t-n}, *for a fixed* n. Predicates p_2, p_3 in Figure 1(a) are examples of the former case. Predicate p_1 is an example of the latter case, since via the decrease/1 and increase/1 predicates it performs tests on the attributes of two consecutive event tuples, i.e. tuples E_{t-n} and E_t with $n = 1$. Unary filters can be evaluated using bounded memory. As a result, a restriction to unary filters is often referred to as the "regular fragment" of CER [22], in similarity to the regular languages, which can also be recognized using bounded memory. In contrast, computational models for ESLs with higher-arity filters go beyond the class of regular automata to families of automata with memory.

Event selection strategies (ESS) and windowing operators. ESS refer to different policies regarding the occurrence of irrelevant events during pattern matching. Prominent ESS are skip-till-next-match and strict-contiguity [39], where the former allows to have irrelevant events (to be "skipped") in between those that explicitly occur in the CER pattern, while the latter does not. As we will show later, our learning method supports both these strategies. Another important CER operator is *windowing*, which specifies a time frame within which a pattern should be matched. We are not concerned with learning windows in this work.

Answer Set Programming. We assume familiarity with ASP and refer to [29] for an in-depth account. An overview of basic ASP constructs that are useful in what follows is provided in the Appendix[2].

Predicate	Meaning
obs$(S_{id}, \text{av}(A, V), T)$	Attribute A has value V in sequence S_{id} at time T.
holds(F, S_{id}, T)	An instance of predicate F is true for sequence S_{id} at time T.
inState(S_{id}, X, T)	An SFA is in state X at the T-th step of processing sequence S_{id}.
transition(S_1, F, S_2)	An SFA moves from state S_1 to state S_2 using the transition guard predicate F.

Table 1: The core predicates used in our ASP encoding.

4 Answer Set Automata

As a first step towards learning SFA-based CE patterns, we present an ASP encoding of such patterns, in programs that we call *answer set automata* (ASA). ASA are executable programs with an one-to-one correspondence to CE patterns and a correctness property, stating that a pattern will be matched against a particular finite piece of input when run with a CER engine, iff its corresponding ASA satisfies a particular query, when run on the same input with an ASP solver. The left part of Table 1 presents the core predicates that we use for our encoding, which we will explain as we go along.

Representing input. To represent a finite input sequence $\mathcal{S}$ of event tuples $E_1, \ldots, E_n$ we use the obs/3 predicate (which stands for "observation"), presented first in Table 1. In particular, we first assign a unique[3] id, s_{id} to the tuple sequence S and then for each tuple $E_t = \langle att_1 = val_1, \ldots, att_m = val_m, time = t \rangle \in \mathcal{S}$ of m attribute-value pairs, we generate m obs/3 atoms of the form obs$(S_{id}, \text{av}(att_i, val_i), T)$. For instance, assuming that the tuple: $\langle necrotic = 110, apoptotic = 420, alive = 770, time = 200 \rangle$ belongs to a sequence $\mathcal{S}$ with $id = s_{id}$, it will be represented by the following facts:
obs$(s_{id}, \text{av}(necrotic, 110), 200)$, obs$(s_{id}, \text{av}(apoptotic, 420), 200)$ obs$(s_{id}, \text{av}(alive, 770), 200)$.
Therefore, an m-attribute/value pair tuple sequence of length n is converted into a *Herbard Interpretation* (set of true ground facts) of $n \times m$ obs/3 facts. In the following we refer to such logical representations of actual input simply as input sequences.

Regarding SFA structure representation, we use integers to denote states. We fix state 0 to always be the start state and use transition/3 facts from Table 1 to denote transitions between states. As an example, Table 2(ii) presents the structure of the SFA from Figure 1, where *any* is a domain constant that evaluates to true.

BK predicates, which are used as building blocks for defining SFA transition rules, are implemented using the holds/3 predicate from Table 1. Table 2(iv) presents an implementation of the BK predicates from Figure 1. For example, the decrease/1 predicate is implemented by using the obs/3 predicate to retrieve and compare consecutive values for A from the input. Given this implementation and the facts obs$(s_{id}, \text{av}(necrotic, 100), 200)$, obs$(s_{id}, \text{av}(apoptotic, 170), 201)$, we can derive the fact holds$(\text{decrease}(alive), s_{id}, 201)$.

The transition guards are defined in terms of the BK predicates, also using the holds/3 predicate. Table 2(iii) presents such definitions for the guard predicates from Figure 1.

The SFA interpreter presented in Table 1 is the core of the encoding, defining the behavior of an SFA. Its first rule simply states

[2] https://cer.iit.demokritos.gr/publications/papers/2023/ecai2023.pdf
[3] Referencing individual input sequences in the ASP encoding will be useful during learning, where such sequences are treated as training examples.

(i) An SFA interpreter:

$\mathsf{inState}(S_{id}, 0, T) \leftarrow \mathsf{sequence}(S_{id}), \mathsf{start}(T).$
$\mathsf{inState}(S_{id}, S2, T+1) \leftarrow \mathsf{inState}(SeqId, S1, T), \mathsf{transition}(S1, F, S2), \mathsf{holds}(F, SeqId, T).$
$\mathsf{accepted}(S_{id}) \leftarrow \mathsf{inState}(S_{id}, X, T), \mathsf{accepting}(X), \mathsf{seqEnd}(S_{id}, T).$

The ASA that corresponds to the SFA from Figure 1

(ii) *Definition of the SFA structure:*
$\mathsf{transition}(0, any, 0).\ \mathsf{transition}(1, p_1, 1).\ \mathsf{transition}(1, p_2, 2).\ \mathsf{transition}(2, p_2, 2).\ \mathsf{transition}(2, p_3, 3).$
(iii) *Transition guards definitions:*
$\mathsf{holds}(p_1, S_{id}, T) \leftarrow \mathsf{holds}(\mathsf{decrease}(alive), S_{id}, T), \mathsf{holds}(\mathsf{increase}(apoptotic), S_{id}, T).$
$\mathsf{holds}(p_2, S_{id}, T) \leftarrow \mathsf{holds}(\mathsf{less\_than\_val}(alive, 800), S_{id}, T).$
$\mathsf{holds}(p_3, S_{id}, T) \leftarrow \mathsf{holds}(\mathsf{less\_than\_att}(alive, necrotic), S_{id}, T).$
(iv) *Background knowledge (BK) predicates definition:*
$\mathsf{holds}(\mathsf{decrease}(A), S_{id}, T) \leftarrow \mathsf{obs}(S_{id}, \mathsf{av}(A, V_1), T), \mathsf{obs}(S_{id}, \mathsf{av}(A, V_2), T-1), V_1 < V_2.$
$\mathsf{holds}(\mathsf{increase}(A), S_{id}, T) \leftarrow \mathsf{obs}(S_{id}, \mathsf{av}(A, V_1), T), \mathsf{obs}(S_{id}, \mathsf{av}(A, V_2), T-1), V_1 > V_2.$
$\mathsf{holds}(\mathsf{less\_than\_val}(A, V), S_{id}, T) \leftarrow \mathsf{obs}(S_{id}, \mathsf{av}(A, V_1), T), V_1 < V.$
$\mathsf{holds}(\mathsf{less\_than\_att}(A_1, A_2), S_{id}, T) \leftarrow \mathsf{obs}(S_{id}, \mathsf{av}(A_1, V_1), T), \mathsf{obs}(S_{id}, \mathsf{av}(A_2, V_2), T), V_1 < V_2.$

Table 2: The core predicates used in our ASP encoding, an SFA interpreter and the implementation of some example BK predicates.

that initially, i.e., at the start point of any sequence, the SFA is in state 0. The second rule states that an SFA moves from S_1 to S_2 at time T if there is a transition-enabling guard F, which evaluates to true at time T. The last rule defines the acceptance condition for a sequence S_{id}, where the seqEnd/2 predicate is properly defined to capture the ending point of the sequence and accepting/1 denotes a designated accepting state.

We may now define an ASA as an ASP program $\Pi = \mathcal{I} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{G}$, where $\mathcal{I}$ is an SFA interpreter, $\mathcal{T}$ is a set of transition/3 facts defining the SFA structure, $\mathcal{B}$ is a set of BK predicate definitions and $\mathcal{G}$ is a set of transition guard definitions. To formally define its transition function, let us first denote by Σ an SFA "alphabet" of obs/3 facts encoding the input and by Q the set of states referenced in $\mathcal{T}$. Note that $\mathcal{T}$ may be seen as defining a mapping $\delta_{\mathcal{T}} : Q \times \mathcal{G} \rightarrow Q$ that maps a state $q_1 \in Q$ and a guard predicate g_{q_1} to a next state q_2, specified by the fact $\mathsf{transition}(q_1, g_{q_1}, q_2) \in \mathcal{T}$[4]. Given such a $\delta_{\mathcal{T}}$ and an ASA Π, we define its transition function $\delta : Q \times 2^{\Sigma} \rightarrow 2^{Q} \cup \{\bot\}$ as:

$$\delta(q, I_t) = \begin{cases} N_q^{t+1} = \{\delta_{\mathcal{T}}(q, g_q) \in Q \mid I_t \cup \Pi \vDash body(p_q)\}, \\ \qquad\qquad \textbf{if } N_q^{+1} \neq \emptyset, \\ \square \in \{\{q\}, \{\bot\}\}, \textbf{ else}. \end{cases}$$

Each I_t should be thought of as being the restriction of an input sequence I to t, i.e. I's subset of obs/3 instances where $T = t$[5]. Given such an I_t and a state q, δ maps q to its set of next states N_q^t, obtained via $\delta_{\mathcal{T}}$, which checks which of q guards' defining conditions (rule bodies) are satisfied by $\Pi \cup I_t$. If N_q^t is empty then the SFA behaves as dictated by a predefined event selection strategy (see Section 3) and it either rejects the input by moving to a "dead state" $\bot$, thus implementing strict-contiguity, or loops on q, following skip-till-next-match. The former strategy is the default for the interpreter from Table 2, since any input S will eventually be rejected – via closed world assumption on accepted/1 – if there exists a point T in S_{id}, such that no $\mathsf{inState}(S_{id}, q, T+1)$ instance can be derived for any state $q \in Q$. In the following section we will also present a way to enforce the skip-till-next-match policy.

Proposition 1 establishes the correctness of our ASA encoding. The proof is provided in the Appendix[6], along with a formal definition of $\mathsf{matches}(P, s)$ for some pattern P and some sequence s.

[4] Note that in the presentation we "overload" the notation of $\mathcal{G}$ to denote both a transition guard predicate g in the definition of $\delta_{\mathcal{T}}$ and its concrete implementation as a rule in $\mathcal{G}$ in the text before the $\delta_{\mathcal{T}}$ definition.

[5] More precisely, I_t should be a segment of I that suffices for evaluating BK predicates. For instance, to evaluate the increase/1, decrease/1 predicates from Table 2 at time t, I_t should contain obs/3 instances corresponding to t and $t-1$.

[6] https://cer.iit.demokritos.gr/publications/papers/2023/ecai2023.pdf

Proposition 1 (Correctness of the ASA encoding) *Let L be any ESL specified by the following grammar:*
$P := \mathsf{FILTER}/1 \mid \mathsf{SEQ}(P_1, P_2) \mid \mathsf{ITER}(P) \mid \mathsf{OR}(P_1, P_2) \mid \mathsf{AND}(P_1, P_2)$.
Let $\mathcal{D}$ be a set of event tuples and $\mathcal{D}^{\star}$ the set of all finite event tuple sequences that may be generated from $\mathcal{D}$. Let P be any L-pattern, whose filters may be expressed as a stratified logic program. Then there is an ASA Π_P, such that for any $s \in \mathcal{D}^{\star}$, $\mathsf{matches}(P, s)$ iff $\mathsf{accepted}(s) \in SM(\Pi_P \cup HI(s))$, where $SM(X)$ denotes the unique stable model of the ASP program X and $HI(s)$ denotes the logical representation of s as a Herband interpretation.

5 Answer Set Automata Learning

We next turn to ASAL, whose core is an abductive learning task implemented as a straightforward application of ASP's generate-and-test methodology, applied on our ASA encoding. Using the representation of an ASA as $\Pi = \mathcal{I} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{G}$, as in Section 4, the goal is to learn its $\mathcal{T}$ and $\mathcal{G}$ parts, from its $\mathcal{I}$ and $\mathcal{B}$ parts, which are provided as input, and a set of training sequences. That is, learn the FSA's structural specification ($\mathcal{T}$ – Table 2(ii)), while synthesizing its transition guard rules ($\mathcal{G}$ – Table 2(iii)). $\mathcal{T}$ is abduced from the ASA interpreter ($\mathcal{I}$ – Table 2(i)) and $\mathcal{G}$ is constructed by abducing BK predicate instances, which may be placed together as conjuncts in the bodies of guard definitions, from $\mathcal{B}$ (Table 2(iv)).

ASAL can learn both deterministic (DSFA) and non-deterministic (NSFA) automata. Although NSFA-based patterns are typically assumed in CER, DSFA-based ones are much easier to interpret and are mandatory in some CER applications [3]. Also, although NSFA are determinazable [11], as in the classical FA case, learnt versions thereof may yield unforeseen behavior, which needs to be manually debugged, in order to extract constraints that can rule-out such behavior in future learning iterations. We thus opt for supporting learning of both types of SFA. Note that the encoding in Section 4 yields NFSA (under the strict-contiguity ESS), since the transition function allows to move to multiple states simultaneously. Enforcing determinism requires additionally to ensure that the outgoing transitions from a state q are guarded by mutually exclusive rules. Providing support for the – commonly assumed in CER – skip-till-next-match ESS, also requires modifications to the encoding. We thus present our approach as targeting DSFA under the skip-till-next-match ESS. Then, obtaining the NSFA setting and the strict-contiguity ESS will only require a simplification of the presented approach.

ASAL is presented in Algorithm 1. It consists of a few simple steps that prepare the *generate* and *test* parts of the encoding (lines 1-3), pass them to an ASP solver to obtain a solution within an optional

(A) Example result of guard_template($n = 3$, $DSFA$ = true, ESS = skip-till-any-match):

(1) holds($g(0,0), S, T$) ← seq(S), time(T), not holds($g(0,1), S, T$), not holds($g(0,2), S, T$).
(2) holds($g(0,1), S, T$) ← holds(body($g(0,1), J$), S, T), not holds($g(0,2), S, T$).
(3) holds($g(0,2), S, T$) ← holds(body($g(0,2), J$), S, T).
(4) holds($g(1,0), S, T$) ← holds(body($g(1,0), J$), S, T), not holds($g(1,2), S, T$).
(5) holds($g(1,1), S, T$) ← seq(S), time(T), not holds($g(1,0), S, T$), not holds($g(1,2), S, T$).
(6) holds($g(1,2), S, T$) ← holds(body($g(1,2), J$), S, T).
(7) holds($g(2,2), S, T$) ← seq(S), time(T).
(8) ← state(S), not transition($S, \_, S$).
(9) holds(body(I, J), S, T) ←
guard(I), disjunct(J), seq(S), time(T), holds(F, S, T) : atom(I, J, F).

(C) Example result of test_part($\mathcal{B}$):

(16) :∼ false_negative(S). [1@0, S]
(17) :∼ false_positive(S). [1@0, S]
(18) :∼ atom(I, J, F). [1@0, I, J, F]
(19) :∼ used_attribute(A). [1@0, A]
(20) used_attribute(A) ← atom(_, _, increase(A)).
(21) used_attribute(A) ← atom(_, _, decrease(A)).
... rest of used_attribute/1 definitions...
(22) false_negative(S) ← pos(S), not accepted(S).
(23) false_positive(S) ← neg(S), accepted(S).

(B) Example result of generate_part($n, m, \mathcal{B}$) for $\mathcal{B}$ from Table 2(iv):

(10) state(0..2). start(0). accepting(2). guard($g(S_1, S_2)$) ← transition($S_1, g(S_1, S_2), S_2$).
(11) {transition($S_1, g(S_1, S_2), S_2$)} ← state(S_1), state(S_2).
(12) {disjunct(1..m)}.
(13) {atom(I, J, increase(A))} ← guard(I), disjunct(J), attr(A).
(14) {atom(I, J, less_than_val(A, V))} ← guard(I), disjunct(J), av(A, V).
(15) {atom(I, J, less_than_att(A_1, A_2))} ← guard(I), disjunct(J), attr(A_1), attr(A_2).

(D) Example of training data:

obs(s_1, av(al, 200), 0), ..., obs(s_1, av(al, 83), 50)
obs(s_1, av(ap, 40), 0), ..., obs(s_1, av(ap, 5), 50)
obs(s_1, av(n, 0), 0), ..., obs(s_1, av(n, 800), 50)
class(s_1, positive)
...
class(s_{10}, negative)

Table 3: Examples of core ASAL components.

Algorithm 1 ASAL($n, m, t, DSFA, ESS, \mathcal{I}, \mathcal{B}, \mathcal{S}$)
Input: n: max number of states ; m: max number of alternative (disjunctive) definitions for a guard; t: solving time limit; $DSFA$: boolean flag for (n-)deterministic SFA; ESS: event selection strategy; $\mathcal{I}$: SFA interpreter; $\mathcal{B}$: BK predicate definitions; $\mathcal{S}$: labeled training set.
Output: $\mathcal{T}$: structural SFA specification of up to n states; $\mathcal{G}$: transition guard definitions

1: $\mathcal{E} \leftarrow$ guard_template($n, DSFA, ESS$).
2: $\mathcal{P}_1 \leftarrow$ generate_part($n, m, \mathcal{B}$).
3: $\mathcal{P}_2 \leftarrow$ test_part($\mathcal{B}$).
4: $\mathcal{M} \leftarrow$ solve($t, \mathcal{E}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{I}, \mathcal{B}, \mathcal{S}$).
5: $(\mathcal{T}, \mathcal{G}) \leftarrow$ assemble($\mathcal{M}, \mathcal{E}$).
6: **return** $(\mathcal{T}, \mathcal{G})$.

7: **function** assemble($\mathcal{M}, \mathcal{E}$):
8: $\mathcal{T} \leftarrow$ all transition/3 facts in $\mathcal{M}$
9: $\mathcal{G} \leftarrow \emptyset$
10: **for each** atom $\alpha \in \mathcal{M}$ of the form $\alpha :=$ atom(i, j, δ):
11: $g_{ij} \leftarrow$ the j-th disjunct of guard i's definition
12: **if** no such g_{ij} exists in $\mathcal{G}$:
13: $\mathcal{G} \leftarrow \mathcal{G} \cup$ holds(g_{ij}, S, T) ← *# adds empty-bodied rule*
14: **else** add δ to the body of g_{ij}
15: **for each** rule $g_{ij} \in \mathcal{G}$
16: add to g_{ij}'s body its corresponding mutual exclusivity conditions specified in $\mathcal{E}$.
17: **return** $(\mathcal{T}, \mathcal{G})$

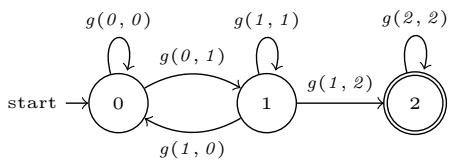
time limit (line 4), and finally interpreting the solution into the result SFA (line 5).

The first of these steps generates a template, i.e., a "skeleton" for the FSA's structure and its guards definitions. The template incorporates a number of design decisions, the first of which is that the starting point for our model is a fully-connected graph of *max_states* nodes, which the SFA induction process then tries to simplify as much as possible by dropping nodes (states) and edges (transitions). An exception is the accepting state, which is always *max_states* (recall that we encode states as integers) and is assumed to be an absorbing one, so it has no outgoing transitions.

An example of such a template for $max\_states = 3$, *DSFA* and skip-till-next-match is presented in rules (1)-(7) of Table 3(A). Assuming that we represent the guard predicate of the (i, j)-transition by $g(i, j)$, rules (1)-(7) provide placeholder definitions for all these predicates that correspond to a fully connected 3-graph, via the holds/3 predicate of the ASA encoding (see Table 1).

Guards corresponding to self-loops on a non-accepting state q (rules (1) and (5)) have no restrictions in their bodies, other than their mutual exclusivity with other outgoing q-guards (recall that the example aims for a DSFA). For instance, rule (1) allows $g(0,0)$ to be trivially satisfied by any event tuple that does not satisfy $g(0,1)$ and $g(0,2)$. The intention is for any such tuple to be effectively "skipped", by triggering a self-loop transition on state 0. The constraint at (8) forces the inclusion of a self-loop for each state referenced in a learnt SFA. This reflects our second design decision, namely that in the case where ESS = skip-till-next-match (as in the example of Table 3(A)), we reserve self-loop transitions for realizing this ESS and delegate the behavior of the ITER operator entirely to learning, to be implemented via cycles between different states.

To conclude the discussion on self loops, note that rule (7) forces the corresponding guard $g(2,2)$ to always be unconditionally satisfied, reflecting the assumption that the accepting state is absorbing.



$g(0,0)$ ← not $g(0,1)$.
$g(1,1)$ ← not $g(1,0)$, not $g(1,2)$.
$g(2,2)$ ← #true.
$g(0,1)$ ← increase($apopt$).
$g(1,0)$ ← less_than_val($apopt$, 700), decrease($alive$), not $g(1,2)$.
$g(1,2)$ ← less_than_att($necr$, $alive$).
$g(1,2)$ ← less_than_val($alive$, 100), increase($apopt$).

Figure 2: A learnt DSFA in simplified form (all predicates stripped of their holds/3 wrapping).

Rules (2), (3), (4) and (6) in the template provide placeholder definitions for "regular" (non self-looping) guards. These rules have an extra condition in their bodies of the form holds(body($g(S_1, S_2), J$), S, T) ($\star$)). Such atoms are meant to serve as placeholders for conjunctions of BK predicate instances. A definition for such placeholder atoms is provided in rule (9), which uses the ASP conditional expression in the end to do exactly that: collect ground instances of BK predicates that are satisfied together, to serve as a conjunctive definition for a guard. For the case of our running example, such BK predicates are generated in rules (13)-(15). J in ($\star$) atoms ranges over the alternative (disjunctive) definitions for $g(S_1, S_2)$ (see rule (12)).

The generate part of Algorithm 1 – see rules (11)-(15) of Table 3(B), uses choice rules to "guess" relevant atoms that when added to the rest of the encoding (i.e the interpreter and the BK) form a working SFA that may be used to accept/reject the training input. The test part of the Algorithm introduces weak constraints that guide the search towards an optimal solution (as defined by the constraints). Rules (16), (17) aim at minimizing the training error, while rules (18), (19) mix-in regularization constraints that try to minimize the

Algorithm 2 ASAL-MCTS(all ASAL args, MBS, ER, MC, $iters$, $roll\_iters$)
Input: MBS : mini-batch size; ER : exploration rate for MCTS; MC : max number of children for a node; $iters$: MCTS iterations; $roll\_iters$: roll-out (simulation iterations) for the default policy.
Output: $\mathcal{T}$, $\mathcal{G}$: as in Algorithm 1

1: $(best\_score, best\_SFA, root.children) \leftarrow (0.0, \emptyset, \emptyset)$
2: expand_node($root$, all ASAL args, MC)
3: **for** $1..iters$ **do**
4: $(score_i, model_i) \leftarrow$ tree_policy($root$, all ASAL args, MC, MBS)
5: **if** $score_i > best\_score$ **then**
6: $(best\_score, best\_SFA) \leftarrow (score_i, best\_score)$
7: $(reward, model) \leftarrow$ default_policy($best\_SFA$, all ASAL args, MC, MBS)
8: propagate_reward($reward$)
return $best\_SFA$

9: **function** expand_node($node_i$, all ASAL args, MC, MBS)
10: **if** $node_i = root$:
11: $\mathcal{D} \leftarrow$ sample_minibatch(MBS)
12: **else**:
13: $\mathcal{D} \leftarrow$ most_urgent_minibatch($node_i$)
14: Run ASAL on $\mathcal{D}$ and keep up to MC locally-optimal solutions SFA_i
15: Evaluate all models in SFA_i on the training set.
16: $node_i.children \leftarrow node.children \cup SFA_i$
17: **return** $(best\_model.score, best\_model)$

18: **function** tree_policy($root$, all ASAL args, MC, MBS)
19: $leaf =$ None
20: With probability p:
21: $leaf \leftarrow$ descent to best leaf
22: **return** expand_node($leaf$, all ASAL args, MC, MBS)
23: With probability $1-p$:
24: **return** expand_node($root$, all ASAL args, MC, MBS)

25: **function** default_policy($node$, all ASAL args, MC, MBS)
26: **for** $1..roll\_iters$:
27: $\mathcal{D} \leftarrow$ sample_minibatch(MBS)
28: Run ASAL on $\mathcal{D}$ and return an optimal solution SFA
29: Evaluate SFA on the training set
30: **return** $(best\_score, best\_model)$ from the roll-out

complexity of the learnt SFM. Symmetry breaking constraints that simplify the solving process are discussed in the Appendix.

Each solution obtained from the solver is interpreted into an SFM by the assemble function of Algorithm 1. This function simply returns the transition/3 atoms found in a solution $\mathcal{M}$, which specify the structure of the SFA, and compiles the guard rules from the atom/3 instances in $\mathcal{M}$, while adding to their bodies the mutual exclusivity conditions dictated by the template, as shown in Algorithm 1. Regarding the latter, note that the template deals with the negation involved in mutual exclusivity in a hierarchical fashion, so that no pair of q-guards $g(q, q_1), g(q, q_2)$ exist that reference each other via negation. This ensures that the program that is compiled by assemble is stratified, which plays a role in the proof of Proposition 1. Figure 2 presents a DSFA that may be learnt from our running example domain. The last two guards are disjunctive alternatives.

To target NSFA and/or strict-contiguity we simply need to remove the mutual exclusivity conditions from the guards' definitions during the template generation and/or remove constraint (8), Table 3.

6 SFA Revision and Monte Carlo Tree Search (MCTS)

ASAL's batch, abductive learning approach can learn an optimal SFA given enough time and memory. The main drawback, however, is that the requirements for such resources grow exponentially with the complexity of the learning task and the size of the input, making the approach infeasible in larger datasets. To address such issues we present an incremental, mini-batch-based version of ASAL. Locally-optimal SFA are continuously revised on new mini-batches, in an effort to approximate a globally adequate solution.

SFA revision aims to specialize or generalize a model, by e.g. adding/removing edges from accepting paths, adding/removing body literals from transition guard rules, or replacing threshold values in such rules with more relaxed/constrained ones. Such revision operations may be realized by the same adbuctive learning process presented in Section 5. By "reversing" ASAL's assemble process from Algorithm 1, an existing SFA may be analyzed into abducible atoms (transition/3 and atom/3 from Table 3). These may be directly injected into the induction program in line 4 of Algorithm 1 as a "prior" and be reasoned upon. The results may be interpreted as a revised version of the initial SFA via the assemble function.

For instance, assume that in the current version of an SFA, guard $g(1,2)$ is defined via two rules $g(1,2) \leftarrow \delta_1$ and $g(1,2) \leftarrow \delta_2, \delta_3$. These correspond to three abducible atoms $\mathsf{atom}(g(1,2),1,\delta_1), \mathsf{atom}(g(1,2),2,\delta_2), \mathsf{atom}(g(1,2),2,\delta_3)$. We may add those into the BK when revising with an new mini-batch, either as facts, or as weak constraints, with the second option allowing such atoms to be removed if deemed useful. If in the generated solution either of these atoms is missing, the rules above need to be generalized accordingly. In contrast, if an additional fact is returned, e.g. $\mathsf{atom}(g(1,2),1,\delta_4)$, then the first of the above rules needs to be specialized into $g(1,2) \leftarrow \delta_1, \delta_4$. Entire guards may be removed by the same process, triggering a restructuring of the SFA with the removal of a transition edge, or new guards – and corresponding edges – may be added.

To implement such a revision-based incremental learning we use MCTS [8]. Algorithm 2 illustrates our implementation of MCTS's *tree* and *default* policies (also called *"roll-out"*, or *"simulation phase"*). Starting from an empty root node we sample a mini-batch and generate a limited number of locally optimal SFA, which are added as children to the root, after evaluated on the training set. Next, for a number of iterations a sequence of the *tree* and *default* policy play-outs take place. During the *tree policy*, the algorithm randomly chooses to either expand the tree horizontally, by descending to the best leaf and expanding it, or vertically, by adding a new child to to the root from a fresh mini-batch sample. In the former case a mini-batch where the leaf node scores poorly (called "most urgent" in Algorithm 2) is selected and used to generate a number of new SFA, which are added as children to the selected leaf. During the roll-out phase the leaf node samples new mini-batches for a number of iterations and generates a new SFA from each. The best score obtained from this sequence of revisions is returned as reward and propagated to the leaf's ancestor nodes. The algorithm keeps track of the best model found so far, which is returned in the end. We use the standard UCT heuristics [8] to select the best child during the *tree policy* phase. All scores (including rewards) are global F_1-scores on the training set.

7 Experimental Evaluation

We evaluate[7] ASAL on 3 CER datasets from the domains of precision medicine, maritime surveillance and activity recognition. The first one contains 644 three-variate sequences of length 50 each, where the attributes correspond to population sizes for *alive*, *necrotic* and *apoptotic* cancer cells. The positive class corresponds to simulations that were deemed promising by human experts.

The *Maritime* dataset was introduced in [3]. It contains data from vessels that cruised around the port of Brest, France. It consists of 5,249 five-variate sequences of length 30, where the attributes are signals for a vessel's *longitude, latitude, speed, heading*, and *course over ground*. The positive class is related to whether a vessel eventually enters the port of Brest.

[7] The code and data are available from https://github.com/nkatzz/asal

	Method	Batch F_1-score	MCTS F_1/iterations 5	MCTS F_1/iterations 10	$\|States\|$	$\|Guards\|$	Grounding (min)	Solving (min)	Total (min)
(A)									
Bio	ASAL	**0.968**			**4**	**5**	1.8	7.2	7.2
	MCTS		0.910	0.962	**4**	7	**0.3**	**0.2**	**3.8**
Maritime	ASAL	**0.982**			**4**	**4**	2.7	12.6	12.6
	MCTS		0.740	0.980	**4**	**4**	**0.3**	**0.1**	**2.8**
Activities	ASAL	**0.788**			**6**	**8**	1.2	18	18
	MCTS		0.740	0.773	7	11	**0.1**	**0.8**	**4.6**
(B)									
Bio	MCTS		0.858	0.968	4	6	0.4	0.9	5.7
Maritime	MCTS		0.915	0.985	5	6	0.6	1.2	7.2
Activities	MCTS		0.740	0.778	7	12	0.2	1.4	7.8
(C)									
Bio	MCTS		0.85	**0.963**	4	6	0.34	0.9	5.3
	RPNI	0.702			13				**0.05**
	EDSM	0.722			12				**0.05**
BioLarge	MCTS		0.852	**0.97**	4	6	0.34	1.02	14.3
	RPNI	Memory error	–	–	–	–	–	–	–
	EDSM	Memory error	–	–	–	–	–	–	–

Table 4: Experimental results.

The *activity recognition* data were obtained from the the CAVIAR dataset[8], consisting of videos of actors performing various activities. The data are annotated at two levels: atomic activities, performed by a single person, e.g. *walking, standing still* and so on, and complex activities, performed by more than one person, e.g. people *meeting* each other (interacting), or *moving* (walking) together. We generated 250 four-variate sequences of length 100 each, each ending in either one of the *meeting* and *moving* complex activities. The features are each person's atomic activities over time, persons' distances and differences in their orientation.

Clingo was used in all experiments. Six BK predicates were used, including equality, attribute/threshold comparisons and attribute/attribute value comparisons. ASAL-MCTS was run with an exploration rate of 0.005 and a max children parameter of 20. Both ASAL versions were run with $max\_states = 6$ and targeted DSFA under skip-till-next-match. In all datasets numerical values were discretized into ten bins using SAX [30]. All experiments were carried-out on a Linux machine with a 3.6GHz processor (4 cores, 8 threads) and 16GB of RAM.

In our first experiment we compared ASAL to ASAL-MCTS. To allow for ASAL to be evaluated we sampled small data fragments from each dataset. Their sizes varied per dataset and were such that ASAL could run in a reasonable amount of time. The sample sizes were 50 examples (each a 4-variate multi-seq.) for CAVIAR, 200 examples (each a 5-variate multi-seq.) for Maritime and 150 examples (each a 5-variate multi-seq.) for Bio. Two ASAL-MCTS runs were performed for each sample, for 5 and 10 iterations respectively, while ASAL-MCTS consumed the data in mini-batches of 20 examples. The process was repeated 5 times. At each iteration a new training fragment was sampled, along with a test set of equal size. The results are presented in Table 4(A) in terms of average testing F_1-scores over the course of the 5 runs, average number of states in the learnt SFA and average size of its guards definitions, average grounding, solving and total training times. Note that in all experiments, the total solving time for ASAL-MCTS corresponds to its 10-iterations run.

The results indicate the ASAL-MCTS was able to effectively match ASAL's performance after 10 iterations. As expected, ASAL-MCTS is significantly more efficient than ASAL. Note that the average total training times for ASAL-MCTS do not reflect the average grounding and solving times, since ASAL-MCTS evaluates a large number of models during a run.

In our second experiment we evaluated ASAL-MCTS's efficacy on whole training sets (rather than samples) in a fivefold cross validation process. In this experiment ASAL-MCTS was run with a batch size of 50. The results are presented in Table 4 and they are similar to those from the previous experiment, with the exception of training times, which slightly elevated, due to the larger batch size used, resulting in larger grounding and solving times. Smaller batch sizes resulted in suboptimal results and required 50 iterations to approximate the results from *batch_size = 10* experiment.

In our final experiment we compared ASAL-MCTS with two classical FA learning algorithms, namely RPNI [34] and EDSM [27], two widely used algorithms of the state-merging (SM) family. These algorithms are quite old, but are still considered SoA in SM-style learning and their LearrnLib[9] implementation used in the experiments is extremely efficient and frequently used by practitioners. The experiment was performed on a univariate version of the bio dataset (which can be handled by RPNI and EDSM), which contains the *alive* attribute only[10]. In this experiment we additionally used a larger version of the bio dataset with 50K simulations, in order to stress-test the compared algorithms' scalability. ASAL-MCTS was used with equality and value comparison BK predicates only, since there are no cross-attribute relations to be discovered. The results are presented in Table 4(C). In the small bio case RPNI and EDSM are lightning-fast, learning a model in approx. three secs. On the other hand, they have significantly inferior predictive performance as compared to ASAL-MCTS. This may be attributed to greedy state merging heuristics. Note that in the large bio dataset both these batch learners terminated with memory errors. In contrast, thanks to its incremental nature, ASAL-MCTS was able to learn a model from this dataset.

8 Conclusion and Future Work

We presented an ASP-based framework for specifying and learning complex event patterns for a particular fragment of the expressivity that is commonly assumed in CER. We also presented an incremental, MCST-based version of our learning approach and evaluated both on three CER datasets, demonstrating empirically their efficacy. Next steps include enhancing the expressive power of the learnt models and further improving scalability.

[8] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/

[9] https://learnlib.de/

[10] This attribute alone is informative enough to learn a useful model in the bio dataset.

Acknowledgements

This work is supported by the project EVENFLOW – *Robust Learning and Reasoning for Complex Event Forecasting*, which has received funding from the European Union's Horizon research and innovation programme under grant agreement No 101070430.

References

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman, 'Efficient pattern matching over event streams', in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 147–160, (2008).

[2] Charilaos Akasiadis, Miguel Ponce-de Leon, Arnau Montagud, Evangelos Michelioudakis, Alexia Atsidakou, Elias Alevizos, Alexander Artikis, Alfonso Valencia, and Georgios Paliouras, 'Parallel model exploration for tumor treatment simulations', *Computational Intelligence*, **38**(4), 1379–1401, (2022).

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

[4] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras, 'Probabilistic complex event recognition: A survey', *ACM Computing Surveys (CSUR)*, **50**(5), 1–31, (2017).

[5] Dana Angluin, 'Learning regular sets from queries and counterexamples', *Information and computation*, **75**(2), 87–106, (1987).

[6] Dana Angluin, Sarah Eisenstat, and Dana Fisman, 'Learning regular languages via alternating automata', in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, (2015).

[7] George Argyros and Loris D'Antoni, 'The learnability of symbolic automata', in *International Conference on Computer Aided Verification*, pp. 427–445. Springer, (2018).

[8] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton, 'A survey of monte carlo tree search methods', *IEEE Transactions on Computational Intelligence and AI in games*, **4**(1), 1–43, (2012).

[9] Andrew Cropper and Sebastijan Dumančić, 'Inductive logic programming at 30: a new introduction', *Journal of Artificial Intelligence Research*, **74**, 765–850, (2022).

[10] Gianpaolo Cugola and Alessandro Margara, 'Tesla: a formally defined event specification language', in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pp. 50–61, (2010).

[11] Loris D'Antoni and Margus Veanes, 'The power of symbolic automata and transducers', in *International Conference on Computer Aided Verification*, pp. 47–67. Springer, (2017).

[12] Colin De la Higuera, *Grammatical inference: learning automata and grammars*, Cambridge University Press, 2010.

[13] Luc De Raedt, *Logical and relational learning*, Springer Science & Business Media, 2008.

[14] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White, 'Towards expressive publish/subscribe systems', in *International Conference on Extending Database Technology*, pp. 627–644. Springer, (2006).

[15] Alan J Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al., 'Cayuga: A general purpose event monitoring system.', in *Cidr*, volume 7, pp. 412–422, (2007).

[16] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom, 'Sase+: An agile language for kleene closure over event streams', *UMass Technical Report*, (2007).

[17] Dana Fisman, Hadar Frenkel, and Sandra Zilles, 'Inferring symbolic automata', *arXiv preprint arXiv:2112.14252*, (2021).

[18] Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo, 'Induction and exploitation of subgoal automata for reinforcement learning', *Journal of Artificial Intelligence Research*, **70**, 1031–1116, (2021).

[19] Lars George, Bruno Cadonna, and Matthias Weidlich, 'Il-miner: instance-level discovery of complex event patterns', *Proceedings of the VLDB Endowment*, **10**(1), 25–36, (2016).

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

[21] Alejandro Grez, Cristian Riveros, and Martín Ugarte, 'A formal framework for complex event processing', in *22nd International Conference on Database Theory (ICDT 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, (2019).

[22] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren, 'A formal framework for complex event recognition', *ACM Transactions on Database Systems (TODS)*, **46**(4), 1–49, (2021).

[23] Nikos Katzouris and Alexander Artikis, 'Woled: a tool for online learning weighted answer set rules for temporal reasoning under uncertainty', in *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 17, pp. 790–799, (2020).

[24] Nikos Katzouris, Georgios Paliouras, and Alexander Artikis, 'Online learning probabilistic event calculus theories in answer set programming', *Theory and Practice of Logic Programming*, **23**(2), 362–386, (2023).

[25] Sarah Kleest-Meißner, Rebecca Sattler, Markus L Schmid, Nicole Schweikardt, and Matthias Weidlich, 'Discovering event queries from traces: laying foundations for subsequence-queries with wildcards and gap-size constraints', in *25th International Conference on Database Theory (ICDT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, (2022).

[26] Sarah Kleest-Meißner, Rebecca Sattler, Markus L Schmid, Nicole Schweikardt, and Matthias Weidlich, 'Discovering multi-dimensional subsequence queries from traces–from theory to practice', *BTW 2023*, (2023).

[27] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price, 'Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm', in *International Colloquium on Grammatical Inference*, pp. 1–12. Springer, (1998).

[28] Yan Li and Tingjian Ge, 'Imminence monitoring of critical events: A representation learning approach', in *Proceedings of the 2021 International Conference on Management of Data*, pp. 1103–1115, (2021).

[29] Vladimir Lifschitz, *Answer set programming*, Springer, 2019.

[30] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi, 'Experiencing sax: a novel symbolic representation of time series', *Data Mining and knowledge discovery*, **15**(2), 107–144, (2007).

[31] Oded Maler and Irini-Eleftheria Mens, 'A generic algorithm for learning symbolic automata from membership queries', in *Models, Algorithms, Logics and Tools*, 146–169, Springer, (2017).

[32] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli, 'Learning from the past: automated rule generation for complex event processing', in *Proceedings of the 8th ACM international conference on distributed event-based systems*, pp. 47–58, (2014).

[33] Stephen H Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad, 'Meta-interpretive learning: application to grammatical inference', *Machine learning*, **94**(1), 25–49, (2014).

[34] José Oncina and Pedro Garcia, 'Identifying regular languages in polynomial time', in *Advances in structural and syntactic pattern recognition*, 99–108, World Scientific, (1992).

[35] Peter R Pietzuch, Brian Shand, and Jean Bacon, 'A framework for event composition in distributed systems', in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 62–82. Springer, (2003).

[36] Miguel Ponce-de Leon, Arnau Montagud, Charilaos Akasiadis, Janina Schreiber, Thaleia Ntiniakou, and Alfonso Valencia, 'Optimizing dosage-specific treatments in a multi-scale model of a tumor growth', *Frontiers in Molecular Biosciences*, **9**, (2022).

[37] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch, 'Distributed complex event processing with query rewriting', in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pp. 1–12, (2009).

[38] Eugene Wu, Yanlei Diao, and Shariq Rizvi, 'High-performance complex event processing over streams', in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 407–418, (2006).

[39] Haopeng Zhang, Yanlei Diao, and Neil Immerman, 'On complexity and optimization of expensive queries in complex event processing', in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 217–228, (2014).