Robust Learning and Reasoning for Complex Event Forecasting

Project Acronym:	EVENFLOW
Grant Agreement Number:	101070430 (HORIZON-CL4-2021-HUMAN-01-01 âĂŞ Research and Innovation Action)
Project full title:	Robust Learning and Reasoning for Complex Event Forecasting

DELIVERABLE

D4.1 – Interim Version of Online Neuro-Symbolic Learning & Reasoning Techniques

Dissemination Level	PU – Public, fully open
Type of Deliverable	R – Document, report
Contractual Date of Delivery:	31 March 2024
Deliverable leader:	NCSR “Demokritos”
Status - version, date:	Final, v1.0, 2024-03-29
Keywords:	Complex Event Recognition and Forecasting, Neuro-symbolic learning and reasoning

Funded by the European Union

This document is part of a project that is funded by the European Union under the Horizon Europe agreement No 101070430. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Commission or the granting authority. The document is the property of the EVENFLOW project and should not be distributed or reproduced without prior approval. Find us at www.evenflow-project.eu.

Executive Summary

We present the progress that has been made during the first half of the project in WP4, regarding neuro-symbolic (NeSy) learning and reasoning techniques for Complex Event Recognition and forecasting (CER/F). We also present our work on exploratory data analysis with the available data in the project, as well as some preliminary results and applications of some of our new techniques in WP4 on such data and additional, surrogate data.

CER/F systems detect, or even forecast ahead of time special events of interest in streaming input. Such events are typically defined via symbolic patterns that correspond to symbolic automata, i.e. finite state machines where the transitions are guarded by logical predicates, as opposed to propositional symbols from a finite alphabet. The symbolic nature of such systems limits their applicability in domains where the input is sub-symbolic, such as sequences of images, or high-dimensional time series. NeSy techniques hold the promise of seamlessly integrating symbolic models, which expect structured input, with neural networks that operate on the perceptual input level. However, NeSy integration in EVENFLOW involves important challenges, since, due to scalability bottlenecks, existing NeSy techniques cannot deal with input of temporal nature.

Regarding the technical progress in WP4, we present a novel technique for scalable NeSy training in temporal domains, which will serve as the backbone for further progress in the project on the NeSy front. The new framework provides a logical/probabilistic language for specifying temporal patterns of interest, which allows to model uncertainty via probabilistic statements. It also provides an algorithm for compiling such patterns into symbolic automata and performing differentiable probabilistic inference with such automata in a scalable fashion. Finally, the new framework provides an interface between the specified symbolic models and neural predictors that map the sub-symbolic input to concrete symbols, allowing to perform NeSy training of neural models, using the symbolic model as a regularizer.

We also present a new method for learning symbolic automata-based complex event patterns from labeled multivariate sequences, representing event traces. Our approach is based on abductive learning in Answer Set Programming and is accompanied by an incremental learning technique, based on Monte Carlo Tree Search, that allow to learn and revise such patterns in a scalable fashion.

Deliverable leader:	NCSR "Demokritos"
Contributors:	Nikos Katzouris, Nikos Manginas, Vasilis Manginas, Georgios Paliouras
Reviewers:	Alessio Lomuscio (ICL), Fatos Gashi (DFKI)
Approved by:	Athanasios Poulakidas, Dimitrios Liparas (INTRA)

Document History:

Version	Date	Contributor(s)	Description
0.1	26/01/2024	Nikos Katzouris	Document skeleton & ToC creation.
0.2	10/02/2024	Nikos Katzouris	Section 3
0.3	2/02/2024	Nikos Katzouris	Section 3, final version
0.4	08/03/2024	Nikos Manginas	Section 2
0.5	08/03/2024	Vasilis Manginas	Section 4
0.6	12/03/2024	Nikos Katzouris	Section 1.
0.7	14/03/2024	Nikos Katzouris, Georgios Paliouras, Nikos Manginas, Vasilis Manginas	Version for internal review.
0.9	28/03/2024	Nikos Katzouris, Nikos Manginas, Vasilis Manginas, Georgios Paliouras	Final version after internal review.
1.0	29/03/2024	Athanasios Poulakidas, Dimitrios Lipars, Nikos Katzouris	QA and final version for submission.

Table of Contents

Executive Summary 1

1 Introduction 7

1.1 Project Information . . . 7

1.2 Document Scope . . . 7

1.2.1 Neuro-Symbolic Complex Event Recognition and Forecasting . . . 7

1.3 Document Structure . . . 10

2 Scalable Neuro-Symbolic Training in Temporal Domains 12

2.1 Introduction . . . 12

2.2 TLog . . . 13

2.2.1 Overview . . . 13

2.2.2 Inference by Knowledge Compilation . . . 14

2.3 DeepTLog . . . 17

2.4 Conclusion . . . 20

2.5 Additional Technical Details . . . 20

2.5.1 PLTLF translation . . . 20

2.5.2 Symbolic automata for probabilistic inference . . . 21

3 Complex Event Pattern Learning 23

3.1 Related Work . . . 24

3.2 Background and Problem Statement . . . 25

3.3 Answer Set Automata . . . 27

3.4 Answer Set Automata Learning . . . 34

3.4.1 Symmetry Breaking Constraints . . . 37

3.5 SFA Revision and Monte Carlo Tree Search (MCTS) . . . 39

3.6 Experimental Evaluation . . . 40

3.6.1 Experiments with EVENFLOW Data . . . 42

3.7 Conclusion and Future Work . . . 42

4 Use Case Data Exploration and Preliminary Experimental Results 43

4.1 Personalized Medicine Use Case . . . 43

4.1.1 Stage Classification . . . 43

4.1.2 Transition Classification . . . 45

4.2 Industry 4.0 Use Case . . . 46

4.3 Infrastructure Lifecycle Assessment Use Case . . . 49

4.3.1 Experimental Setup and Data . . . 49

4.3.2 Classification Task . . . 50

4.3.3 Regression Task . . . 50

5 Conclusions and Future Work 52

5.1 End-to-end Verification of Neuro-Symbolic Systems 52

List of Figures

1 A typical baseline approach for handling sub-symbolic input in the CER/F domain. The input in this example consists of sequences of images from an on-board robot camera in the DFKI use case in EVENFLOW. 8
2 Neuro-symbolic Complex Event Recognition & Forecasting. 10
3 A sample program for matching a sequential pattern over a sequence of digits sampled from a given distribution 13
4 A sample TLOG program (a) and its corresponding Dynamic Bayesian network (b). The program is first checked for correctness and is then translated to an intermediate logic which is then compiled to an automaton. 15
5 A very simple TLOG program (a) and the corresponding automaton it has been compiled to (b). Some transitions are missing from (b) for brevity. They are $t_{0\to1}$ which is $(d\wedge\neg p_0)\vee(d\wedge\neg p_2)\vee(p_0\wedge\neg r)\vee(r\wedge\neg p_0\vee(p_2\wedge r\wedge\neg d)$ and $t_{0\to2}$ which is $(d\wedge\neg p_2)\vee(d\wedge\neg r)\vee(p_0\wedge\neg r)\vee(p_1\wedge\neg r)\vee(p_2\wedge r\wedge\neg d)\vee(r\wedge\neg p_0\wedge\neg p_1)$. The abbreviation r stands for rain d for delay p_0 for the probability of the first rule, p_1 the second and so on. Note that these functions are not represented in this form but rather compactly as SDDs. 16
6 The conversion of a TLOG program to a DEEPTLOGPROGRAM. The language now uses `#ext` probabilities which can be given by external means in each timestep and most notably by neural networks. 18
7 CER for tumor progression simulation optimization; **(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 cocktail over the course of a simulation [2, 52]. . . 24
8 A learnt DSFA in simplified form (all predicates stripped of their holds/3 wrapping). 36
9 The trace of a robot moving around the smart factory floor. 46
10 An excerpt of time series robot mobility data illustrating the relevant features. . 48
11 An illustrative symbolic automaton learnt with ASAL from DFKI data. 48

List of Tables

1 Benchmark for the compilation of synthetic programs. For each setting 50 random programs were produced. $|V|$ represents the number of variables in the program. I is the number of interface variables and we also show the ratio, i.e. how many more states are necessary in the approach taken by [58] 16

2 Results on MNIST sequence task for different sequence lengths. We report the accuracy achieved on the test set as well as the time for training per epoch. We train for a total of 50 epochs for both systems with the same learning rate. Best values per metric (secs/epoch, accuracy) in the table are in bold. 19
3 The core predicates used in our ASP encoding. 28
4 The core predicates used in our ASP encoding, an SFA interpreter and the implementation of some example BK predicates. 28
5 Examples of core ASAL components. 34
6 Experimental results. 40
7 Distribution of breast cancer patients chosen from the TCGA-BRCA dataset . . 43
8 Stage classification with gene expression input 44
9 Stage classification with enriched pathway input 45
10 Distribution of stage transitions for breast cancer patients chosen from the TCGA-BRCA dataset 46
11 Transition classification with enriched pathway input 47
12 Leakage binary classification given input from all ten sensors 50
13 Distance from each sensor to the leakage point 51
14 Leakage distance regression given input readings from one sensor. Each cell includes the minimum mean average error (MAE) achieved over all models tested, as well as the amount of time ”spanned” by one feature vector in that configuration of parameters. 51

Definitions, Acronyms and Abbreviations

ASAL	Answer Set Automata Learning
ASP	Answer Set Programming
BK	Background Knowledge
CER/F	Complex Event Recognition & Forecasting
CE	Complex Event
DBN	Dynamic Bayesian Network
DeepProbLog	Deep Probabilistic Logic Programming
DeepStochLog	Deep Stochastic Logic Programming
DSFA	Deterministic Symbolic Finite Automaton
ESL	Event Specification Language
ESS	Event Selection Strategy
MCTS	Monte Carlo Tree Search
MSO	Monadic Second Order Logic
NeSy	Neuro-symbolic
NSFA	Non-Deterministic Symbolic Finite Automaton
PLTL$_F$	Pure-past Linear Temporal Logic over finite traces
PMC	Pattern Markov Chain
SDD	Sentential Decision Diagrams
SE	Simple Event
SFA	Symbolic Finite Automaton
TNF	Tumor Necrotic Factor

1 Introduction

1.1 Project Information

EVENFLOW is developing hybrid learning techniques for complex event forecasting, which combine deep learning with logic-based learning and reasoning into neuro-symbolic forecasting models. The envisioned methods combine (i) neural representation learning techniques, capable of constructing event-based features from streams of perception-level data with (ii) symbolic learning and reasoning tools, that utilize such features to synthesize high-level, interpretable patterns of critical situations to be forecast.

Crucial in the EVENFLOW approach is the online nature of the learning methods, which makes them applicable to evolving data flows and allows to utilize rich domain knowledge that is becoming available progressively. To deal with the brittleness of neural predictors and the high volume/velocity of temporal data flows, the EVENFLOW techniques rely on novel, formal verification techniques for machine learning, in addition to a suite of scalability algorithms for federated training and incremental model construction. The learnt forecasters will be interpretable and scalable, allowing for fully explainable insights, delivered in a timely fashion and enabling proactive decision making.

EVENFLOW will be evaluated on three use cases related to (i) oncological forecasting in precision medicine, (ii) safe and efficient behaviour of autonomous transportation robots in smart factories and (iii) reliable life cycle assessment of critical infrastructure.

1.2 Document Scope

This document presents the advancements made in WP4 within the scope of EVENFLOW in the first 18 months of the project. WP4 in EVENFLOW focuses on the development of neuro-symbolic (NeSy) learning and reasoning techniques in temporal domains, towards building event recognition and forecasting systems that are able to operate on sub-symbolic, perceptual input and integrate symbolic predictive models with neural networks. This deliverable elaborates on the generic tools developed and the way they are applied in real world scenarios also derived from EVENFLOW use cases. The current deliverable will further evolve in the second half of the project to "D4.2 Final Version of Online Neuro-Symbolic Learning & Reasoning Techniques" delivered on Month 36 of EVENFLOW.

1.2.1 Neuro-Symbolic Complex Event Recognition and Forecasting

Complex Event Recognition [32] and Forecasting [3] (CER/F) systems seek to detect, or even forecast ahead of time, occurrences of special events of interest, across a set of input data streams. The input streams consist of *simple events*, which are time-stamped pieces of information, and the output are the detected/forecast instances of the target situations, which are called *complex*

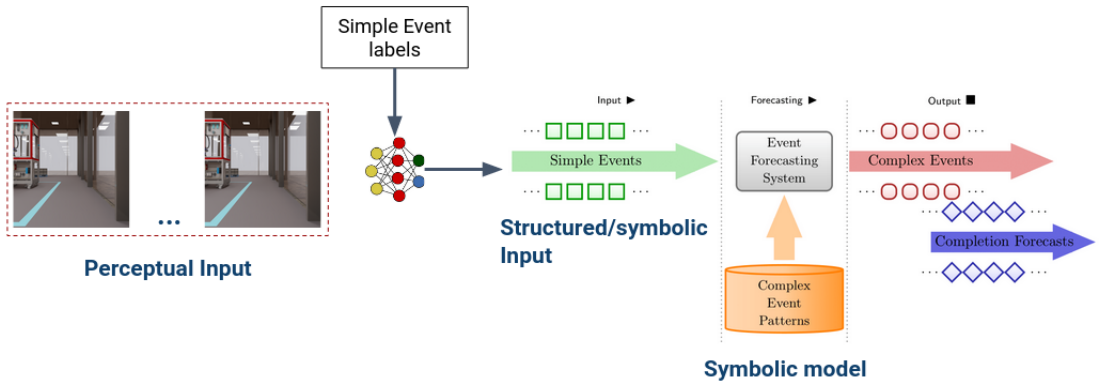


Figure 1: A typical baseline approach for handling sub-symbolic input in the CER/F domain. The input in this example consists of sequences of images from an on-board robot camera in the DFKI use case in EVENFLOW.

events and are usually defined as spatio-temporal combinations of the simple events.

To perform the recognition process, CER/F systems rely on a set of complex event patterns, which are declarative specifications of the interesting situations to be monitored. Such situations usually involve sets of correlated events that are expected to occur in a sequential fashion. The formalisms that are used to define such patterns, called *event specification languages* [34], allow to compose such patterns from some basic operators, which typically extend classical regular expression constructs, such as sequencing and Kleene closure, with additional operators, such as filtering, i.e. satisfiability testing of logical predicates against the temporal input. Due to the sequential nature of such complex event patterns, the corresponding computational objects are a special kind of automata (finite state machines), called *symbolic automata* [14], whose transitions are guarded by predicates (corresponding to the filtering operators), rather than by mere symbols from a finite alphabet. The recognition process then amounts to matching such automata-based patterns against the simple event input, i.e. reaching an accepting state in the automaton during processing the input stream, while the forecasting task amounts to deriving probabilistic estimates of future full pattern matches from partial matches that have been observed so far.

Regarding the forecasting task, in particular, which is a core aspect of the EVENFLOW approach, the actual automaton pattern is first converted into a *Pattern Markov Chain* (PMC), a probabilistic model that allows to reason about the behavior of the initial pattern. A PMC is a Markov Chain obtained from the pattern itself, by associating with each transition the probability of executing that transition, estimated empirically from an initial segment of the input stream. From the PMC, using standard Markov Chain machinery, it is possible to compute the so-called *waiting time distributions* [3]. For any given non-accepting state q in a pattern, its waiting time is a random variable defined as the number of transitions required for the pattern to reach an accepting state (i.e. a full match) from q. Therefore, the waiting-time distributions allow to answer queries like "what is the probability of a pattern completion from state q in n steps into the future".

The symbolic nature of existing CER/F systems restricts their applicability to symbolic

input. However, numerous applications deal with sub-symbolic, perceptual level input, such as sequences of images, or high-dimensional time series. A typical baseline approach in such cases is to train a neural predictor to map the sub-symbolic input to a set of symbols, corresponding to the simple events in our case, which are then passed to the symbolic model that handles the downstream CER/F task, as illustrated in Figure 1. Such approaches are sub-optimal, since the neural predictors are trained in isolation, ignoring the downstream task. They are also often infeasible, since they require large amounts of simple event-labeled data, which are usually difficult to obtain.

On the other hand, the alternative of abandoning symbolic techniques altogether and resorting to purely neural techniques instead comes with important shortcomings. These include the back-box nature of neural models, their brittleness, their poor generalization in out-of-distribution settings and their lack of support for incorporating existing domain knowledge. The latter is of particular significance, since in the CER/F domain it is often the case that domain experts are interested in monitoring known complex event patterns across the input data streams. Therefore, CER/F approaches are required to both support expressive event specification languages that allow to manually define known patterns and computational tools that interface such languages with sub-symbolic input.

Neuro-symbolic (NeSy) learning and reasoning techniques seek to bridge the gap between neural and symbolic techniques by combining the best of two worlds. A variety of NeSy techniques have been proposed in the literature – we refer to [48] for a comprehensive survey.

Figure 2 illustrates the NeSy approach to CER/F that we pursue in EVENFLOW. The neural model is not trained in isolation as in the case illustrated in Figure 1, but it receives a learning signal that takes into account the symbolic model's performance on the downstream task. This is achieved by using some form of differentiable inference with the symbolic model, in order to produce the predictions/forecasts. This allows to compute the gradients directly on the symbolic model. In turn, this entails that the gradients carry some form of semantic information related to specific points of failure w.r.t. the symbolic model and, therefore, the network updates during gradient back-propagation, progressively align the neural model's predictions with the symbolic model's requirements.

The general schema that is illustrated in Figure 2 points to the relatively recent family of NeSy interfaces, which typically use the symbolic model as a regularized for the neural model. In such approaches the neural and the symbolic parts are clearly separated, and they communicate via some gradient-based interface. This increases the transparency of such approaches, as opposed for instance to approaches that embed the logic into the network, using e.g. fuzzy logic [8]. Also, such NeSy interfaces may be easily implemented using off-the-shelf tools (solvers, reasoners) and they allow for a formal probabilistic semantics, since differentiable inference is usually achieved via some probabilistic variant of crisp logical inference. The latter is not the case in NeSy approaches that rely on logical embeddings. Yet, it is an important requirement in EVENFLOW, since, as discussed above, existing event forecasting techniques rely on such

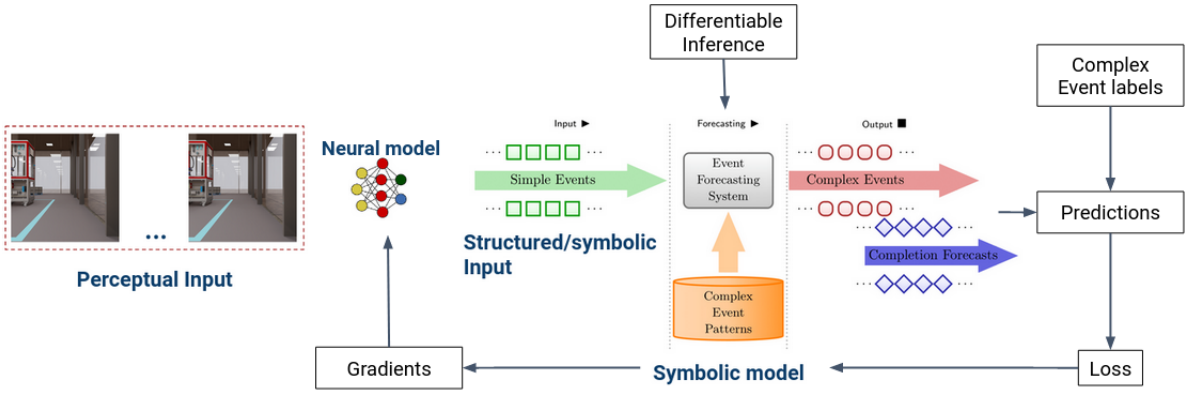


Figure 2: Neuro-symbolic Complex Event Recognition & Forecasting.

probabilistic semantics to estimate the likelihood of future full pattern matches.

For the above reasons we focus on such NeSy interfaces in EVENFLOW. However, this comes with important challenges. As will be made clear in Section 2, existing, state-of-the-art NeSy approaches of the logic-as-regularizer family face important scalability barriers that make them impractical in temporal domains.

To address this issue, we propose a novel neural/probabilistic NeSy learning and reasoning framework that offers a language for specifying complex event patterns, which are compiled into symbolic automata at runtime. The new framework comes with a formal probabilistic semantics, which allows for differentiable probabilistic inference with the compiled automata and results in a scalable NeSy training procedure, particularly tailored for temporal domains.

Interesting complex event patterns are not always known beforehand, while existing ones often need to be revised in response to changes in the input data characteristics. Therefore, machine learning techniques for inducing such patterns from data are necessary. However, as will be made clear in Section 3.4, existing approaches for learning symbolic automata structures have several limitations, which make them impractical in EVENFLOW. To address this issue, we propose a novel structure learning technique for symbolic automata, capable of learning and revising such patterns from labeled event traces.

1.3 Document Structure

This document consists of the following chapters:

- Chapter 2 presents our novel neural/probabilistic framework for scalable NeSy learning and reasoning in temporal domains.
- Chapter 3 presents our novel symbolic automata learning and revision technique, based on Answer Set Programming.
- Chapter 4 presents our work on exploratory analysis of the available EVENFLOW use case data and some preliminary results.

- In Chapter 5 we summarize our work, present directions for future work and conclude.

2 Scalable Neuro-Symbolic Training in Temporal Domains

2.1 Introduction

Hybrid systems which are capable of combining neural networks with symbolic components are attracting significant attention. Many such systems [46, 61, 59, 57, 56] are based on logic programming, interpreted however under probabilistic semantics. This approach allows to create compositional systems. Information is extracted from perceptive inputs via neural networks towards uncertain facts. These facts are then reasoned upon with a logic program, optionally itself incorporating uncertainty. However, current systems are not viable (in terms of scalability) for temporal applications, which are of course plentiful.

DeepProbLog [46] extends the probabilistic programming language Problog [23] with the concept of a neural predicate. With it, probabilistic facts in a program can be externally computed, via a neural network, by observing some perceptive input, e.g. an image. With this simple concept Problog has been upgraded towards a hybrid system which enjoys the benefits of a powerful probabilistic language with clear semantics, which is also capable of reasoning over complex high-dimensional inputs. Crucially, the two components are integrated via a probabilistic framework, which allows to train the neural component with weak supervision.

Additional DeepProbLog-inspired systems have been proposed including [55, 59]. Their aim is to improve scalability while retaining rich probabilistic semantics. However, all such systems struggle to scale to real world temporal applications. Perhaps the most fundamental reason for this is their general purpose-nature, with which scalability issues are bound to arise.

To fill this gap, we propose TLOG, a simple logical/probabilistic language to specify dynamical systems and present its neurosymbolic integration. While Problog is a language that allows to succinctly represent complex Bayesian Networks, TLOG focuses on Dynamic Bayesian Networks instead. Wanting to borrow the state of the art inference techniques of Problog based on knowledge compilation, one can look to [58] for knowledge compilation techniques for dynamical models. However, the technique presented there, while scaling impressively compared to traditional exact inference techniques for DBNs, still suffers from large scalability issues, mainly due to an exponential blowup in the interface variables (those which define temporal dependencies). We therefore resort to a novel inference scheme for TLOG which compiles a program into symbolic automata[14]. Our initial implementation of the translation from TLOG programs to symbolic automata, while naive, shows that it is possible to perform exact inference for TLOG programs even when the corresponding DBNs would not look amenable to exact inference using current state of the art approaches.

Inference on the resulting symbolic automaton can be performed differentiably. The language TLOG is therefore extended to DEEPTLOG trivially using techniques similar to [46, 59, 61]. We summarize our contributions:

- We design the language TLOG which is based on Dynamic Bayesian Networks.

```
0.8: even.
{ 0.2: smaller_than_3; 0.5: between_3_and_6; 0.3: larger_than_6 }.

t01 ← even, larger_than_6.
t12 ← not even, not larger_than_6.
t23 ← smaller_than_3.

q0 ← q0[-], not t01.
q1 ← q0[-], t01.
q1 ← q1[-], not t12.
q2 ← q1[-], t12.
q2 ← q2[-], not t23.
q3 ← q2[-], t23.
q3 ← q3[-].

accept ← q3.
```

Figure 3: A sample program for matching a sequential pattern over a sequence of digits sampled from a given distribution

- We create a novel inference procedure for TLOG , and thus also DBNs, based on symbolic automata, which scales favourably to previous approaches [58].
- We extend TLOG towards DEEPTLOG which integrates the core language with the neural predicate [46] and present synthetic applications to showcase our scalability improvements.

2.2 TLog

2.2.1 Overview

TLOG is a probabilistic logic programming language similar to Problog [23]. It targets dynamic, instead of static, probabilistic models and temporal reasoning. A TLOG program consists of facts and rules optionally embellished with probabilities. A fact is given by $\mathtt{p} : \mathtt{a}$ meaning that atom $\mathtt{a}$ is true with probability $\mathtt{p}$ at each timestep. A rule is given by $\mathtt{p} : \mathtt{h} \leftarrow \mathtt{l}_1, \ldots, \mathtt{l}_n$ where $\mathtt{l}_i$ is a literal, i.e. an atom or its negation. If the body of a rule is satisfied at timestep t then its head will also be true at timestep t with probability $\mathtt{p}$. The construct $\mathtt{a}[-]$ is used to refer to the value of $\mathtt{a}$ one timestep before, $\mathtt{a}[--]$ two timesteps and so on. Such lookback atoms can be used in the bodies of rules. For example, the rule $0.2 : \mathtt{a} \leftarrow \mathtt{b},\ \mathtt{not}\ \mathtt{a}[-]$ reads if $\mathtt{b}$ is true in the current timestep and $\mathtt{a}$ was not true in the previous one, then with probability 0.2 $\mathtt{a}$ will be true in this timestep. Since the semantics of the language are given over DBNs, cycles are disallowed (this is mostly for ease of implementation). The language supports annotated disjunctions to allow specifying distributions over categorical variables. A sample TLOG program is given in Figure 3 and a simpler program along with its DBN is shown in Figure 4

One should note that every TLOG program can be already expressed in Problog or in Stochastic Definite Clause Grammars (the reasoning backbone of DeepStochLog [59]). The novelty of TLOG lies in its inference procedure which is tailored to temporal tasks and will be

shown to scale considerably better. Since TLOG is equivalent to the above languages it follows that all TLOG programs define a distribution over the variables which appear in it and the logic is interpreted over the distribution semantics [53].

2.2.2 Inference by Knowledge Compilation

Inference in TLOG programs is based on the principles of knowledge compilation [16, 11]. The core idea of probabilistic inference by knowledge compilation is to convert a probabilistic model to a logical theory and then compile this theory to an arithmetic circuit on which queries can be answered effectively. While the compilation procedure might be very costly, if it is successful, this cost can be amortised over multiple queries.

The idea of inference in TLOG is similar, however, instead of compiling to arithmetic circuits, which are inherently acyclic structures, a TLOG program is compiled to an automaton. Perhaps the work closest to our inference procedure is [58] which introduced knowledge compilation for inference in DBNs. The algorithm there worked via the external aid of recursion, effectively providing this notion of cyclicity that seems necessary for a practical inference algorithm in temporal applications. However, the algorithm still incurred an exponential blowup, rendering it infeasible for DBNs with multiple interface variables. To the best of our knowledge, we are the first to suggest compiling dynamic probabilistic models to automata for exact and scalable temporal probabilistic inference.

The first step of inference in TLOG is compilation. For brevity we only give a very high level overview of the compilation procedure and the reader is referenced to Section 2.5 for more information. We also note that we believe this compilation procedure is suboptimal and that a much more performant compiler is possible, however our current approach is simple to implement and allows for quick development of a prototype system.

A TLOG program is first converted to the temporal logic PLTLF. This is the first notable difference between the compilation procedure of TLOG vs probabilistic logical languages not focused on dynamical models, e.g. Problog. The knowledge base used to represent the TLOG model is not based on propositional logic but rather on one of its temporal variants. The PLTLF encoding is then converted to an automaton via external tools. The complexity of converting PLTLF to an automaton is EXPTIME. Whether this is tight for TLOG programs is an issue for further investigation, since we only use a subset of the full logic. Once an automaton is computed it is then further processed by converting its guards to Sentential Decision Diagrams [15]. If the compilation is successful, the automaton can be evaluated and many probabilistic queries over the TLOG program can be answered tractably. A simple model along with its automaton is shown in Figure 5.

We next answer the first question of this work. **Can TLOG programs be compiled to automata efficiently?** We first give a bit of background. As already mentioned, the compilation procedure of TLOG programs first reduces to an intermediate logic, namely the pure past version of LTLF. The problem of converting LTLF formulas to automata has been studied extensively

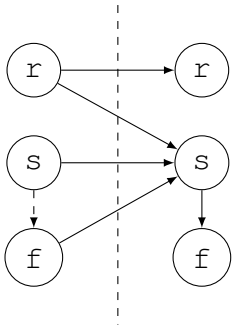
```
0.5: rain.
0.8: rain ← rain[-].

0.05: sensor(faulty).
0.30: sensor(faulty) ← rain[-].

sensor(faulty) ←
    sensor(faulty)[-], not fix[-].

0.70: fix ← sensor(faulty).
```

(a)



(b)

Figure 4: A sample TLOG program (a) and its corresponding Dynamic Bayesian network (b). The program is first checked for correctness and is then translated to an intermediate logic which is then compiled to an automaton.

[20, 63, 9, 18, 19] and many improvements have been made. Our initial implementation interfaces with one of the older systems for the automaton compilation from a logical formula. Since then, the field has advanced significantly and current implementations scale to much more complex formulas. Unfortunately interfacing with newer systems is technically challenging and we leave it for future work.

Secondly, as already mentioned, previous research for compiling Dynamic Bayesian Networks, on which TLOG is based, scaled exponentially in the number of interface variables. Interface variables are those with outgoing edges to the next time slice in a DBN. Equivalently, for TLOG programs they are variables which occur in the bodies of rules and have an offset operator, e.g. `fix`$[-]$ renders `fix` an interface variable. The previous approach was therefore impractical for models with more than a handful of interface variables.

We show results of the compilation procedure of TLOG in Table 1 on randomly generated programs.

Table 1: Benchmark for the compilation of synthetic programs. For each setting 50 random programs were produced. $|V|$ represents the number of variables in the program. I is the number of interface variables and we also show the ratio, i.e. how many more states are necessary in the approach taken by [58]

$\|V\|$	#clauses	Average			Maximum			Solved(%)
		$\|I\|$	#states	ratio	$\|I\|$	#states	ratio	
30	100	17	710	175	21	632	3318	100
35	120	20	1380	546	24	2756	6010	88
35	140	21	1880	9290	24	7693	2181	92
40	130	21	2055	1223	25	1672	20068	92
40	150	23	3023	3734	28	3389	79207	84
45	150	25	2604	9290	28	1242	213277	72
45	180	27	3540	37912	30	1720	624268	32

Our conclusion is that the compilation of TLOG programs is promising and despite its naivety the current compiler is capable of solving problems beyond the capabilities of what was possible for exact inference in DBNs. Models with up to 30 interface variables are able to be compiled by our new technique, something which was out of reach previously. In practice it seems that for medium size programs even a naive compilation is often sufficient.

The next question is how one can perform inference given that an automaton has been compiled for a given TLOG program. While a number of queries can be answered, we will provide an algorithm for computing marginals, since this is the most useful query in our neurosymbolic integration. However, we note that any query answerable by an arithmetic circuit satisfying determinism and decomposability [11] is achievable for TLOG programs once an automaton has been constructed.

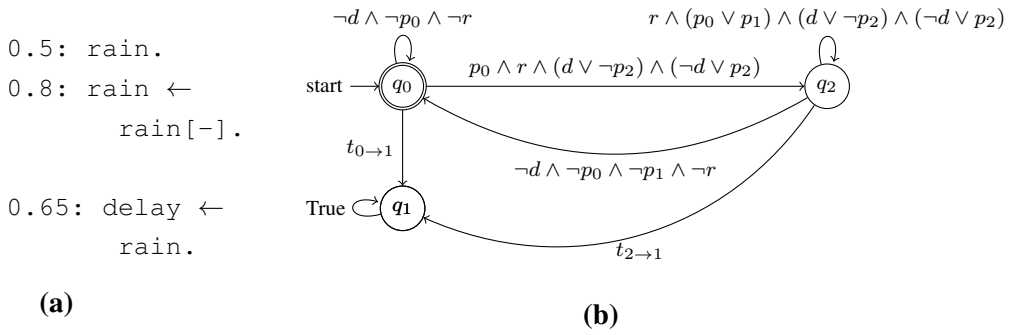


Figure 5: A very simple TLOG program (a) and the corresponding automaton it has been compiled to (b). Some transitions are missing from (b) for brevity. They are $t_{0\to1}$ which is $(d \wedge \neg p_0) \vee (d \wedge \neg p_2) \vee (p_0 \wedge \neg r) \vee (r \wedge \neg p_0 \vee (p_2 \wedge r \wedge \neg d)$ and $t_{0\to2}$ which is $(d \wedge \neg p_2) \vee (d \wedge \neg r) \vee (p_0 \wedge \neg r) \vee (p_1 \wedge \neg r) \vee (p_2 \wedge r \wedge \neg d) \vee (r \wedge \neg p_0 \wedge \neg p_1)$. The abbreviation r stands for rain d for delay p_0 for the probability of the first rule, p_1 the second and so on. Note that these functions are not represented in this form but rather compactly as SDDs.

All automata we have been discussing are deterministic, i.e. for a given assignment to the automaton's symbols there is only one transition outgoing from each state that will be true. If this condition is not met our approach is impossible.

Marginal computation in TLOG follows a very similar approach as done for standard arithmetic circuits in static probabilistic models. In each timestep the probability of the guards of the automaton are computed, then via matrix multiplications the probability of being in each state is updated. Once the sequence has been consumed the probability of accepting the sequence is computed (this is always 1 for TLOG programs that have not been conditioned since they always define a normalized distribution similarly to Problog) and the partial derivatives of each variable per timestep are calculated. These values are the marginals and are used for the bulk of tasks which we use TLOG for. For the program above this computation would for example give for the value of `delay` the vector $[0.325, 0.455, 0.507, 0.528, 0.536]$ for the first 5 timesteps.

This process can be parallelized efficiently due to the continuous matrix multiplications for which great support exists and is also completely differentiable. If the probability of rain each day was not fixed to 0.5 but was instead given externally via a neural network one can compute the marginals of the program and retain the gradient from the neural network though the inference procedure of the symbolic component.

We therefore ask the second question of the paper **Can inference in compiled TLOG programs be done efficiently and differentiably?** We answer affirmatively.

2.3 DeepTLog

The focus of this work is not only the language TLOG and its inference procedure but also on its neurosymbolic integration. To achieve this, we add a new syntax to the language. Wherever a probability was specified in a TLOG program in DEEPTLOG the probability can be replaced by $\#$`ext` followed by a number. The program from Figure 3 can therefore be converted to the DEEPTLOG program in Figure 6

```
#ext1: even.
{ #ext2: smaller_than_3; #ext2: between_3_and_6; #ext3: larger_than_6 }.

t01 ← even, larger_than_6.
t12 ← not even, not larger_than_6.
t23 ← smaller_than_3.

q0 ← q0[-], not t01.
q1 ← q0[-], t01.
q1 ← q1[-], not t12.
q2 ← q1[-], t12.
q2 ← q2[-], not t23.
q3 ← q2[-], t23.
q3 ← q3[-].

accept ← q3.
```

Figure 6: The conversion of a TLOG program to a DEEPTLOGPROGRAM. The language now uses #ext probabilities which can be given by external means in each timestep and most notably by neural networks.

TLOG now expects that a vector will be given for each #ext probability which signifies the probability of the fact being true in each timestep. As aforementioned the inference procedure for TLOG is differentiable and therefore one can for example compute the probability of accept in each timestep and as long as the values provided for #ext probabilities are attached with a gradient then so will the marginals. These can therefore be supervised and the gradients backpropagated back to the neural network allowing us to train via weak supervision. This lift from TLOG to DEEPTLOG is very similar to DeepProbLog and DeepStochLog.

How does DEEPTLOG compare with existing neurosymbolic system from its family?

We create a synthetic benchmark to stress test DEEPTLOG. We observe a sequence of MNIST images and the task is to match a temporal pattern on the digits. At each timestep a neural network is given an image of a digit and predicts the probability of the digit being even as well as its magnitude in three different classes, i.e. $x \leq 3$, $3 < x \leq 6$, $x > 6$. For example, an image representing the digit 6 should be maped to even and $3 < x \leq 6$. We then wish to at some point in the sequence see a digit which is even and $x > 6$ then eventually a different digit which is odd and $x \leq 6$ and then eventually a digit that is $x \leq 3$. This is a very simple regular expression-like pattern similar to those often encountered in Complex Event Recognition tasks [31]. The system is trained via weak supervision. No labels are provided on the latent concepts, e.g. even. Instead, the program is run on the probabilities given by the neural networks and the probability of the pattern matching is computed. This value is then supervised, i.e. whether the whole sequence matches or not and the loss back-propagated to the neural networks. Data are therefore given in pairs where each pair is a sequence of image digits along with a label indicating

whether the synthetic pattern holds in the sequence. We do not compare against DeepProbLog [46] as it has already been shown that in such sequential tasks it is quickly outscaled by its sibling DeepStochLog [59]. We therefore only compare against the newer system.

Sequence Length	DeepStochLog		DeepTLog	
	s/epoch	accuracy	s/epoch	accuracy
10	**53**	82	60	**86**
15	80	85	**43**	**90**
20	140	63	**34**	**91**
25	188	75	**34**	**91**
30	230	72	**27**	**97**
40	364	-	**19**	**93**

Table 2: Results on MNIST sequence task for different sequence lengths. We report the accuracy achieved on the test set as well as the time for training per epoch. We train for a total of 50 epochs for both systems with the same learning rate. Best values per metric (secs/epoch, accuracy) in the table are in bold.

As can be seen from the results in Table 2, our system scales much more efficiently on the task. It can process much longer sequences without this having an adverse effect on scalability. This is important since in real world applications sequence sizes can be large with long range dependencies. Once DEEPTLOG models have been compiled to automata they can be efficiently evaluated regardless of sequence size. This is not the case for DeepStochLog which generates proof trees that are dependent on its current sequence and grow large. For our system it is possible to process sequences up to 10000 timesteps which is far beyond what DeepStochLog can achieve. As aforementioned the system DeepProbLog is omitted since its scalability is even worse than DeepStochLog based on [59] as well as our own experimentation. We note that for each setting the number of sequences generated is smaller. The same budget of training images exists and as sequence length grows the amount of sequences that can be generated decreases which explains why the runtime of DEEPTLOG is smaller for larger sequence lengths.

In summary, the performance of DEEPTLOG stays consistent with increasing sizes while the performance of DeepstochLog seems to deteriorate as sequence length increases. This drop in performance is not well understood since both systems should provide similar learning signals. However, the large difference in running times is to be expected. What is truly the bottleneck in temporal applications is scalability and even from this simple benchmark we can see that our system scales considerably better in temporal domains. As the complexity of problems grows it is only expected this difference will be increased as long as complex TLOG models can be compiled. Note that when comparing DeepStochLog with DeepProbLog the focus was again on scalability as in most works concerning this sort of systems.

2.4 Conclusion

We proposed a simple knowledge representation language for temporal applications TLOG . We designed a new inference procedure for TLOG which is based on symbolic automata. Since TLOG programs are dynamic bayesian networks this inference procedure is of much interest independently. Our inference procedure, while initially naive, was shown to scale beyond what was possible with current state of the art inference technique for DBNs. We then presented the neurosymbolic integration of TLOG DEEPTLOG which even on simple benchmarks is shown to considerably outperform its competition.

2.5 Additional Technical Details

Some more technical details on PLTLF translation and on symbolic automata follow.

2.5.1 PLTLF translation

A simple algorithm for compiling a TLOG program to a symbolic automaton, is to first go through some other temporal logic. While more efficient compilation procedures could be possible this approach allows building a prototype system very quickly. For technical reasons, having to do with existing compilers, we choose to go through the modal logic for finite traces PLTLF which is a variant of LTLF where modal operators refer to the past. From PLTLF we only use two modal operators ($\ominus$ and $\boxminus$) in English *before* and *historically* respectively. The translation is quite straightforward. Given a probabilistic fact `p : a` we give the corresponding formula $\boxminus(a \leftrightarrow p_\theta)$ where p_θ is a newly introduced logical variable. Annotated disjunctions are handled by encoding constraints as given in [11] for handling categorical variables. For a rule $\texttt{p} : \texttt{h} \leftarrow \texttt{l}_1, \ldots, \texttt{l}_\texttt{n}$ we give the translation $\boxminus(a \leftrightarrow p_\theta \wedge l_1 \wedge \cdots \wedge l_n)$ where p_θ is again a newly introduced variable. If some literal also includes a lookback e.g. $\texttt{l}[-]$ then it is converted to $\ominus l$. For instance, the rule $\texttt{0.2} : \texttt{a} \leftarrow \texttt{b},\ \texttt{not a}[-]$ is converted to $\boxminus(a \leftrightarrow p_\theta \wedge b \wedge \neg(\ominus a))$. If multiple rules have the same head, then the right-hand side of the implication is a disjunction. The program is then the conjunction of all such clauses. Finally, TLOG allows for specifying variables which are initially true. For instance, `q0` for Program 3 can be initially asserted as true. This capability is given to the user externally from the language itself. The PLTLF encoding of Program 3 with `q0` asserted to be initially true is:

$$\boxminus(\texttt{smaller\_than\_3} \leftrightarrow \texttt{p1})$$
$$\boxminus(\texttt{between\_3\_and\_6} \leftrightarrow \texttt{p2})$$
$$\boxminus(\texttt{larger\_than\_6} \leftrightarrow \texttt{p3})$$
$$\boxminus(\texttt{even} \leftrightarrow \texttt{p4})$$
$$\boxminus(\texttt{t01} \leftrightarrow \texttt{even} \wedge \texttt{larger\_than\_6})$$
$$\boxminus(\texttt{t12} \leftrightarrow \neg\texttt{even} \wedge \neg\texttt{larger\_than\_6})$$
$$\boxminus(\texttt{t23} \leftrightarrow \texttt{smaller\_than\_3})$$
$$\boxminus(\texttt{q0} \leftrightarrow ((\ominus\texttt{q0}) \wedge \neg\texttt{t01}) \vee (\neg \ominus \texttt{true}))$$
$$\boxminus(\texttt{q1} \leftrightarrow ((\ominus\texttt{q0}) \wedge \texttt{t01}) \vee (\ominus\texttt{q1} \wedge \neg\texttt{t12}))$$
$$\boxminus(\texttt{q2} \leftrightarrow ((\ominus\texttt{q1}) \wedge \texttt{t12}) \vee (\ominus\texttt{q2} \wedge \neg\texttt{t23}))$$
$$\boxminus(\texttt{q3} \leftrightarrow ((\ominus\texttt{q2}) \wedge \texttt{t23}) \vee (\ominus\texttt{q3}))$$
$$\boxminus(\neg\texttt{p1} \vee \neg\texttt{p2})$$
$$\boxminus(\neg\texttt{p1} \vee \neg\texttt{p3})$$
$$\boxminus(\neg\texttt{p2} \vee \neg\texttt{p3})$$
$$\boxminus(\texttt{p1} \vee \texttt{p2} \vee \texttt{p3})$$

Such encodings can be given to an external tool in order to be compiled to a symbolic automaton. We use [28] for this purpose.

2.5.2 Symbolic automata for probabilistic inference

The automaton produced by the compilation of PLTLF is then converted to a special form in which the transition guards are compiled to Sentential Decision Diagrams [15]. Note here that one could use BDDs or sd-DNNF circuits here and this may in fact be considerably more efficient. For advantages and disadvantages of these structures the reader is referred to [16, 15]. One the automaton has been converted to this form, i.e. its transitions are compiled to SDDs one can perform probabilistic inference in response to a number of queries, e.g. marginals or MPEs.

For our purposes, is suffices that the reader understands the following difference. While, in a classical finite automaton a transition $(q_0) \xrightarrow{a} (q_1)$ occurs on reading a symbol e.g. a from an alphabet, in a symbolic automaton transitions take the form: $(q_0) \xrightarrow{\neg a \vee b} (q_1)$ .Multiple symbols can be read at each timestep of the automaton run and transitions can carry arbitrary logical expressions over a structured alphabet. Symbolic automata underpin many implementations of temporal logics on finite inputs, e.g. LTLF[20], PLTLF[17], MSO [35].

It should be clear that as long as TLOG programs correspond to DBNs and since DBNs can

be converted to HMMs whose logical component is itself an automaton [38], then it must be that the logical component of TLOG programs is indeed an automaton.

3 Complex Event Pattern Learning

Complex Event Recognition (CER) systems [32, 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) [34]. Despite the diversity of such languages, a minimal event processing operators that every ESL should support [62, 4, 32, 33] 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 [14] (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 [60, 1, 62, 26, 24, 25, 54, 51, 13, 3]. Prominent areas of CER research, then, concern the study of trade-offs between ESLs' expressive power and pattern matching complexity [26, 34], 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 [37, 36], and less on the sequential nature of such events; others do support operators such as iteration [47, 39, 40, 30], or filtering predicates [42, 29], 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) [43], 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.

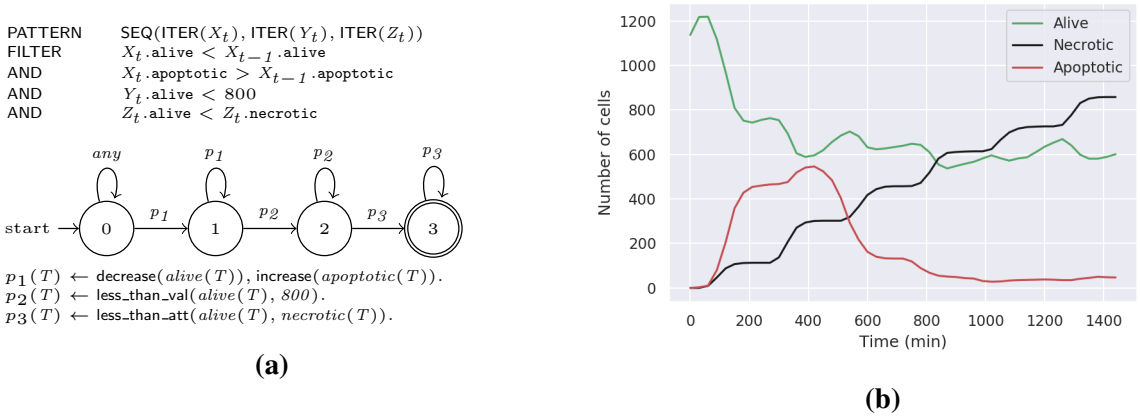


Figure 7: CER for tumor progression simulation optimization; **(a)** A CE pattern that captures the simulation on the right (upper), its corresponding SFA (middle) and the SFA's guards (bottom); **(b)** Temporal evolution of different cell populations after the injection of a drug cocktail over the course of a simulation [2, 52].

3.1 Related Work

The methods introduced in [30] and [47] 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 [39, 40], 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 [40]. Moreover, this technique is also restricted to a purely sequential ESL. The method of [42] 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 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.

ASAL supports the core CER operators of *sequence*, *filtering* and *iteration*, thus going beyond the sequence pattern learning task of [30, 47, 39, 40]. Moreover, in contrast to [42], 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 [21] has a long history in the literature [5, 50, 41, 6, 29]. Most existing techniques are either noise-intolerant leaners [5, 6], or rely on greedy heuristics for *state merging* [50, 41], 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 [45, 7, 27], 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) [12, 22] since its early days. More recent ILP frameworks have also been applied to the task [49, 29]. Both these approaches are designed to learn from small univariate training samples and cannot deal with noisy input.

3.2 Background and Problem Statement

We begin with a brief description of a tumor evolution simulation optimization task [2, 52], which we will refer to as a running example throughout the paper. Figure 7(b) presents the temporal evolution of tumor cell populations of different types (*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 [52].

Event tuples. Typically, CER systems operate on streams of *event tuples* [32, 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 7(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 7(a) presents an example of such a pattern that may be matched

against the input of the simulation in Figure 7(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 cases such as those presented in the simulation of Figure 7(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 7(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 3.3). 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

[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.

that such language bias is currently restricted to unary predicates, such as $p_1/1, p_2/1, p_3/1$ in Figure 7(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 7(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 [34], 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 [62], 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. In what follows we assume familiarity with ASP and refer to [43] for an in-depth account. In this section we review some basic ASP constructs that will be useful in what follows. Throughout, we use the Clingo[2] syntax for representing ASP expressions. A choice rule is an expression of the form $\{\alpha\} \leftarrow \delta_1, \ldots, \delta_n$, with the intuitive meaning that whenever the body $\delta_1, \ldots, \delta_n$ is satisfied by an answer set I of a program that includes the choice rule, instances of the head α are arbitrarily included in I (satisfied) as well. A weak constraint is an expression of the form $:\sim \delta_1, \ldots, \delta_n.[w@p, t_1, \ldots, t_k]$, where δ_i's are literals, called the body of the constraint, w and p are integers, called respectively the *weight* and the *priority level* of the constraint and $t_1, \ldots, t_k$ are ASP terms. A grounding/instance of a weak constraint c is an expression that results from c by replacing all variables in $\delta_1, \ldots, \delta_n, t_1, \ldots, t_k$ by constants. Such an instance is satisfied by an answer set I_Π of a program Π that includes c if I_Π satisfies c's ground body, which incurs a penalty of w on I_Π. I_Π's total cost is the sum of penalties resulting from each instance of c that is satisfied by I_Π. Inclusion of weak constraints in an ASP program triggers an optimization process that yields answer sets of minimum cost. Priority levels in weak constraints model the constraints' relative importance, since the aforementioned optimization process attempts to first minimize the total cost due to weak constraints of higher priority levels.

3.3 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

[2] https://potassco.org/

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

Table 3: The core predicates used in our ASP encoding.

(i) An SFA interpreter:

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

The ASA that corresponds to the SFA from Figure 7

(ii) ***Definition of the SFA structure:***
$\mathsf{transition}(0, any, 0).$ $\mathsf{transition}(0, p_1, 1).$ $\mathsf{transition}(1, p_1, 1).$ $\mathsf{transition}(1, p_2, 2).$ $\mathsf{transition}(2, p_2, 2).$ $\mathsf{transition}(2, p_3, 3).$
(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 4: The core predicates used in our ASP encoding, an SFA interpreter and the implementation of some example BK predicates.

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 3 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 3. 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 $\mathsf{obs}(S_{id}, \mathsf{av}(att_i, val_i), T)$. 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:
$\mathsf{obs}(s_{id}, \mathsf{av}(necrotic, 110), 200)$, $\mathsf{obs}(s_{id}, \mathsf{av}(apoptotic, 420), 200)$ $\mathsf{obs}(s_{id}, \mathsf{av}(alive, 770), 200)$.
Therefore, an m-attribute/value pair tuple sequence of length n is converted into a *Herbrand 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.

[3]Referencing individual input sequences in the ASP encoding will be useful during learning, where such sequences are treated as training examples.

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 3 to denote transitions between states. As an example, Table 4(ii) presents the structure of the SFA from Figure 7, 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 3. Table 4(iv) presents an implementation of the BK predicates from Figure 7. 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 $\mathsf{obs}(s_{id}, \mathsf{av}(necrotic, 100), 200)$, $\mathsf{obs}(s_{id}, \mathsf{av}(apoptotic, 170), 201)$, we can derive the fact $\mathsf{holds}(\mathsf{decrease}(alive), s_{id}, 201)$.

The transition guards are defined in terms of the BK predicates, also using the holds/3 predicate. Table 4(iii) presents such definitions for the guard predicates from Figure 7.

The SFA interpreter presented in Table 3 is the core of the encoding, defining the behavior of an SFA. Its first rule simply states 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 $\delta_{\mathcal{T}} : Q \times \mathcal{G} \to Q$ 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 $\delta : Q \times 2^{\Sigma} \to 2^{Q} \cup \{\bot\}$ as:

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

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 3.2) and it either rejects the input by moving to a "dead state" $\bot$, thus

[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 4 at time t, I_t should contain obs/3 instances corresponding to t and $t-1$.

implementing strict-contiguity, or loops on q, following skip-till-next-match. The former strategy is the default for the interpreter from Table 4, 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 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 below establishes the correctness of our ASA encoding. We precede the proof with a formal definition of a matching between a pattern and an input sequence. This definition is used by Proposition 1.

Definition 1. *Let $\mathcal{D}$ be a set of event tuples, $\mathcal{D}^\star$ the set of all finite event tuple sequences that may be generated from $\mathcal{D}$, with ϵ denoting the empty sequence, and L be any Event Specification Language $\mathcal{L}$, specified by the following grammar:*

$$P := \textit{FILTER}/1 \mid \textit{SEQ}(P_1, P_2) \mid \textit{ITER}(P) \mid \textit{OR}(P_1, P_2) \quad (\star)$$

where FILTER/1 denotes unary filters. We define the following relation matches $\subseteq \mathcal{L} \times \mathcal{D}^\star$, inductively on the structure of $\mathcal{L}$, as follows:

- *If $P := \textit{FILTER}/1$, then matches$(P, s) \Leftrightarrow \exists\, e \in \mathcal{D}^\star : s = e \;\&\; \textit{FILTER}(e)$, i.e. s is a single-tuple sequence, which satisfies the conditions defined by FILTER. Note that the fact that s needs to be of length 1 – single-tuple sequence – follows from the fact that FILTER predicates are unary).*
- *If $P := \textit{SEQ}(P_1, P_2)$, then*
  matches$(P, s) \Leftrightarrow \exists s_1, u, s_2 \in \mathcal{D}^\star : s = s_1 \cdot u \cdot s_2$ & matches(P_1, s_1) & matches(P_2, s_2), where "$\cdot$" denotes concatenation. Note that this definition complies with the skip-till-next-match selection strategy. For strict-contiguity we need to require that $u = \epsilon$.
- *If $P := \textit{ITER}(P_1)$, then matches$(P, s) \Leftrightarrow \exists\, u \in \mathcal{D} : s = u^+ \;\&$ matches(P_1, u), where for any event tuple sequence s, $s^+ = \bigcup_{n \geq 1} s_n$ and s_n is the concatenation of s with itself n times.*
- *If $P := \mathsf{OR}(P_1, P_2)$, then matches$(P, s) \Leftrightarrow$ matches(P_1, s), or matches(P_2, s).*

We may now proceed to Proposition 1.

Proposition 1 (Correctness of the ASA encoding)**.** *Let L be any ESL specified by the following grammar: $P := \textit{FILTER}/1 \mid \textit{SEQ}(P_1, P_2) \mid \textit{ITER}(P) \mid \textit{OR}(P_1, P_2) \mid \textit{AND}(P_1, P_2)$. 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 $s \in \mathcal{D}^\star$, matches(P, s) iff 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.*

Proof. Note first that any ASA with a stratified set of filters predicates is stratified, since the rest of the ASA encoding in the paper is negation-free. Therefore, any ASA A with stratified filters has a unique stable model $SM(X)$.

The proof proceeds by induction on the structure of P. With the exception of the base case, for the other three cases the induction hypothesis is that the proposition holds for the sub-patterns of the initial pattern. In each case we will construct an ASA as an ASP program $\Pi = \mathcal{I} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{G}$, with a fixed ASA interpreter $\mathcal{I}$:

$$\begin{aligned}
&\mathsf{inState}(S_{id}, 0, T) \leftarrow \mathsf{sequence}(S_{id}), \mathsf{start}(T).\\
&\mathsf{inState}(S_{id}, S2, T+1) \leftarrow \mathsf{inState}(S, S1, T), \mathsf{transition}(S1, F, S2), \mathsf{holds}(F, S_{id}, T).\\
&\mathsf{acceptedAt}(S_{id}, T) \leftarrow \mathsf{inState}(S_{id}, X, T), \mathsf{accepting}(X).\\
&\mathsf{accepted}(S_{id}) \leftarrow \mathsf{acceptedAt}(S_{id}, T), \mathsf{seqEnd}(S_{id}, T).
\end{aligned} \tag{2}$$

and varying transition transition/3 facts $\mathcal{T}$, BK predicate definitions $\mathcal{B}$ and transition guard definitions $\mathcal{G}$. Note that we slightly teak the definition of the ASA interpreter from that presented in Table 4(i), by adding an extra rule that records the time points at which the automaton is in an accepting state, which will be useful in the proof, and defining the acceptance condition as being in an accepting state at the end of the sequence. In what follows we will use the symbol of an input sequence as its *id*, so we write e.g. $\mathsf{accepted}(s)$ to denote that sequence s is accepted. We may now proceed to the inductive proof.

- **Base case**. Let $s \in \mathcal{D}^\star$ and $P := \textsc{filter}/1$. Assume that $\mathsf{matches}(P, s)$. Then, s is single tuple that satisfies the filter. Define a two-state SFA as an ASA program Π with:

$$\begin{aligned}
\mathcal{T} &= \{\mathsf{transition}(0, g(0,1), 1),\ \mathsf{start}(0),\ \mathsf{accepting}(1)\}\\
\mathcal{B} &= \{p/1\}\\
\mathcal{G} &= \{g(0,1) \leftarrow p(\langle (a_1^s, v_1^s), \ldots, (a_n^s, v_n^s)\rangle).\}
\end{aligned} \tag{3}$$

  where $p/1$ is an ASP implementation of $\textsc{filter}/1$. In particular, we assume that $p/1$ operates on the tuple of s's attribute/value pairs, $\langle (a_1^s, v_1^s), \ldots, (a_n^s, v_n^s)\rangle$ and performs the operations indicated by $\textsc{filter}/1$ on these pairs. Also, since s is a single-tuple (i.e. a sequence of length 1), its logical representation as a Herbrand interpretation will be a set of observation facts indexed by a single time point (0), of the form:

$$HI(s) = \{\mathsf{obs}(s, \mathsf{av}(a_1^s, v_1^s), 0), \ldots \mathsf{obs}(s, \mathsf{av}(a_n^s, v_n^s), 0)\} \tag{4}$$

  Since FILTER/1 satisfies s, it follows that $HI(s) \vDash p(\langle (a_1^s, v_1^s), \ldots, (a_n^s, v_n^s)\rangle$ and, therefore, the body in the following instantiation of $\mathcal{I}$'s second rule is satisfied by $\Pi \cup HI(s)$:

$$\mathsf{inState}(s,1,1) \leftarrow \mathsf{inState}(s,0,0), \mathsf{transition}(0,g(0,1),1), \mathsf{holds}(g(0,1),s,0).$$

It follows that the automaton is in its accepting state at time point 1, which is the end of the input sequence, therefore $\mathsf{accepted}(s) \in SM(\Pi \cup HI(s))$. Conversely, if the above holds, then s is a single-tuple sequence satisfied by FILTER, therefore, $\mathsf{matches}(P,s)$.

- **Case 2:** $P := \mathsf{SEQ}(P_1,P_2)$ under strict-contiguity. Assume that $\mathsf{matches}(P,s)$. Then $s = s_1 \cdot s_2$, i.e. it is the concatenation of tuple sub-sequences s_1 and s_2, of length t_1 and t_2 respectively, each of which satisfies the induction hypothesis. Let Π_1 and Π_2 be the corresponding ASA. Define an SFA whose states are the union of states in Π_1 and Π_2 as an ASA program Π_3 with:

$$\begin{aligned}
\mathcal{T}(\Pi_3) &= \mathcal{T}(\Pi_1) \cup \mathcal{T}(\Pi_2) \cup \{\mathsf{transition}(accepting(\Pi_1), g, start(\Pi_2))\} \\
\mathcal{B}(\Pi_3) &= \mathcal{B}(\Pi_1) \cup \mathcal{B}(\Pi_2) \cup \top \\
\mathcal{G}(\Pi_3) &= \mathcal{G}(\Pi_1) \cup \mathcal{G}(\Pi_2) \cup \{g \leftarrow \top\}
\end{aligned} \tag{5}$$

where $accepting(\Pi_1)$ is Π_1's accepting state, $start(\Pi_2)$ is Pi_2's start state and g is a new, trivially satisfied transition guard (a fact indicated by its definition, $g \leftarrow \top$).

The logical form of s as a Herbrand interpretation will be

$$HI(s) = HI(s_1)^{0..t_1} \cup HI(s_2)^{t_1+1..t_1+t_2+1}$$

where the superscripts denote the time points that index the observation facts in each interpretation. Since, by the construction of Π_3 and the definition of $HI(s)$ it holds that $SM(\Pi_1 \cup HI(s_1)) \subset SM(\Pi_3 \cup HI(s))$, and from the inductive hypothesis we have that $\mathsf{acceptedAt}(s_1,t_1) \in SM(\Pi_1 \cup HI(s_1))$, it follows that $\mathsf{acceptedAt}(s_1,t_1) \in SM(\Pi_3 \cup HI(s))$, and since t_1 is the end-point of s_1, it follows that $\mathsf{accepted}(s_1) \in SM(\Pi_3 \cup HI(s))$, therefore, $\mathsf{inState}(s, accepting_{\Pi_1}, t_1) \in SM(\Pi_3 \cup HI(s))$. Since the newly-introduced in Π_3 guard g is trivially satisfied, it follows from the extra transition fact in Π_3 and the interpreter's second rule that $\mathsf{inState}(s, start(\Pi_2), t_1+1) \in SM(\Pi_3 \cup HI(s))$. Now, since from the inductive hypothesis we have that $\mathsf{acceptedAt}(s_2, t1+t_2+1) \in SM(\Pi_2 \cup HI(s_2)^{t1+t_2+1})$ and $SM(\Pi_2 \cup HI(s_2)^{t1+t_2+1}) \subset SM(\Pi_3 \cup HI(s))$, it follows that $\mathsf{acceptedAt}(s_2, t1+t_2+1) \in SM(\Pi_3 \cup HI(s))$ and, therefore, $\mathsf{accepted}(s) \in SM(\Pi_3 \cup HI(s))$.

Conversely, if the above holds, then there must be sub-sequences s_1 and s_2, such that $s = s_1 \cdot s_2$ and additionally, $\mathsf{matches}(P_1,s_1)$ & $\mathsf{matches}(P_2,s_2)$, where P_1 and P_2 are

$\mathcal{L}$-patterns that correspond to programs Π_1 and Π_2, and these patterns exist from the inductive hypothesis. Therefore, it holds from Definition 1 that $\mathsf{matches}(\mathsf{SEQ}(P_1, P_2), s)$.

- **Case 3:** $P := \mathsf{ITER}(P_1)$. Let $s \in \mathcal{D}^\star$ and assume that $\mathsf{matches}(P, s)$. Then, from Definition 1 we have that there exists a $u \in \mathcal{D}$ such that $s = u^+$ & $\mathsf{matches}(P_1, u)$, where $u^+ = \bigcup_{n\geq 1} s_n$ with u_n being the concatenation of u with itself n times[6]. Therefore, $s = u_1 \cdot u_2 \cdot \cdots \cdot u_n$, with $u_1 = u$, $u_2 = u \cdot u$ and so on until u_n. Given the ASA Π that accepts u from the inductive hypothesis, it is straightforward to extend it to an ASA Π', which accepts u^+, by adding a new start and accepting state and an extra transition guarded by a trivially satisfied guard (as in case 2 of the proof) between the existing accepting state and the existing start state. In more detail, define Π' as follows:

$$\begin{aligned}
\mathcal{T}(\Pi') &= \mathcal{T}(\Pi) \cup \{\mathsf{transition}(\mathit{accepting}(\Pi), g, \mathit{start}(\Pi)),\\
&\qquad\qquad \mathsf{transition}(\mathit{start}(\Pi'), g, \mathit{start}(\Pi)),\\
&\qquad\qquad \mathsf{transition}(\mathit{accepting}(\Pi), g, \mathit{accepting}(\Pi'))\} \qquad (6)\\
\mathcal{B}(\Pi') &= \mathcal{B}(\Pi) \cup \top\\
\mathcal{G}(\Pi') &= \mathcal{G}(\Pi) \cup \{g \leftarrow \top\}
\end{aligned}$$

where $\mathit{accepting}(\Pi)$ and $\mathit{start}(\Pi)$ are the accepting/start states of the inductive hypothesis ASA Π, $\mathit{accepting}(\Pi')$ and $\mathit{start}(\Pi')$ are the new accepting and start states and g is a trivially satisfied guard as before. Note that this is a non-deterministic ASA, which simultaneously moves to $\mathit{start}(\Pi)$ and $\mathit{accepting}(\Pi')$ from $\mathit{accepting}(\Pi)$, i.e. it accepts a sub-sequence accepted by the inductive hypothesis ASA and moves back to its start state to continue processing the next sub-sequence. If $t_s = \sum_{n=1}^{n} u_n$ is the length of s it follows that $\mathsf{acceptedAt}(s, t_s) \in SM(\Pi' \cup HI(s))$ and therefore $\mathsf{accepted}(s) \in SM(\Pi' \cup HI(s))$.

  Conversely, assume that the above holds. Then s must necessarily be a concatenation of u's, each of which is accepted by Π, which has a corresponding $\mathcal{L}$-pattern from the inductive hypothesis. Therefore, s must satisfy the $\mathsf{ITER}(P)$ part of Definition 1 1.

- **Case 4:** $P := \mathsf{SEQ}(P_1, P_2)$ under skip-till-next-match. This is a combination of $P := \mathsf{SEQ}(P_1, P_2)$ under strict-contiguity and $P := \mathsf{ITER}(P_1)$. Indeed, in this case $s = s_1 \cdot u \cdot s_2$ with u non-empty and $u \neq s_1$, $u \neq s_2$. The pattern that matches s is then $\mathsf{SEQ}(P_{s_1}, \mathsf{ITER}(P_u), P_{s_2})$ and the proof is a combination of cases 2 and 3 above.

- **Case 5:** $P := \mathsf{OR}(P_1, P_2)$. This case is straightforward, by introducing a new accepting state reachable by a trivially satisfied guard from the accepting states of the inductive hypothesis automata.

[6]Note that the definition of iteration/Kleene Closure that we use in this paper is a repetition of a pattern one or more times, differently from the definition that is sometimes used, of a repetition of zero or more times

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

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

(B) Example result of generate_part$(n, m, \mathcal{B})$ for $\mathcal{B}$ from Table 4(iv):

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

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

(16) $:\sim$ false_negative(S). $[1@0, S]$

(17) $:\sim$ false_positive(S). $[1@0, S]$

(17) $:\sim$ false_positive(S). $[1@0, S]$

(18) $:\sim$ atom(I, J, F). $[1@0, I, J, F]$

(19) $:\sim$ used_attribute(A). $[1@0, A]$

(20) used_attribute$(A) \leftarrow$ atom$(_, _,$ increase$(A))$.

(21) used_attribute$(A) \leftarrow$ atom$(_, _,$ decrease$(A))$.
... rest of used_attribute/1 definitions...
(22) false_negative$(S) \leftarrow$ pos(S), not accepted(S).
(23) false_positive$(S) \leftarrow$ neg(S), accepted(S).

(D) Example of training data:

obs$(s_1,$ av$(al, 200), 0), \ldots,$ obs$(s_1,$ av$(al, 83), 50)$

obs$(s_1,$ av$(ap, 40), 0), \ldots,$ obs$(s_1,$ av$(ap, 5), 50)$
obs$(s_1,$ av$(n, 0), 0), \ldots,$ obs$(s_1,$ av$(n, 800), 50)$

class$(s_1,$ positive$)$

class$(s_1,$ positive$)$
...

class$(s_{10},$ negative$)$

Table 5: Examples of core ASAL components.

- **Case 6:** $P := \text{AND}(P_1, P_2)$. This case is a straightforward combination of sequence and disjunction, since accepting the conjunction of two patterns P_1 and P_2 amounts to either accepting P_1 and then P_2, or P_2 and then P_1.

□

3.4 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 3.3, 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 4(ii)), while synthesizing its transition guard rules ($\mathcal{G}$ – Table 4(iii)). $\mathcal{T}$ is abduced from the ASA interpreter ($\mathcal{I}$ – Table 4(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 4(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 inter-

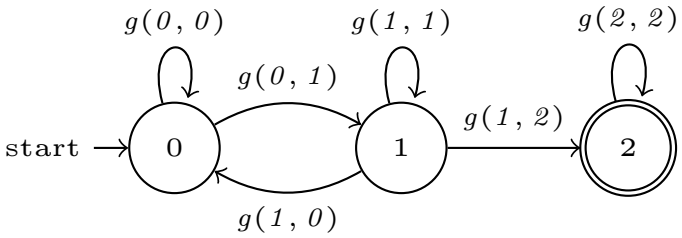
Algorithm 1 ASAL$(n, m, t, DSFA, ESS, \mathcal{I}, \mathcal{B}, \mathcal{S})$
Input: n: max number of states; m: max number of alternative (disjunctive) definitions for a guard; t: solving time limit; $DSFA$: boolean flag for (n-)deterministic SFA; ESS: event selection strategy; $\mathcal{I}$: SFA interpreter; $\mathcal{B}$: 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$(g_{ij}, S, T) \leftarrow$ *# 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** $(\mathcal{T}, \mathcal{G})$

pret and are mandatory in some CER applications [3]. Also, although NSFA are determinazable [14], 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 3.3 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 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)



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

Figure 8: A learnt DSFA in simplified form (all predicates stripped of their holds/3 wrapping).

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 5(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 3).

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 5(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.

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 5(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 complexity of the learnt SFM. Symmetry breaking constraints that simplify the solving process are presented in Section 3.4.1.

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 8 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 5.

3.4.1 Symmetry Breaking Constraints

Constraints (7) and (8) below impose an ordering on the used states, so that new states are introduced on demand only when previous states have already been already used. This avoids symmetric solutions based on a re-ordering of the states where e.g. a transition between states 1 and 2 in one solution automaton may be "replaced" by a transition between states 1 and in another. Note that state 1 is always the start state.

$$\begin{aligned} \leftarrow\ & \mathsf{state}(S), S \neq 1, S \neq 2, \mathsf{not\ accepting}(S), \\ & \#\mathsf{count}\{S1 : \mathsf{state}(S1), S1 = S - 1, S1 \neq 1\} = 0. \end{aligned} \tag{7}$$

$$\begin{aligned} \leftarrow\ & \mathsf{transition}(I, f(I, K), K), \mathsf{state}(J), \\ & I < J, J < K, \mathsf{not\ transition}(I, f(I, J), J). \end{aligned} \tag{8}$$

Constraint (9) below imposes an ordering on the conjunction *id*'s to also avoid symmetric solutions, where the same guard appears multiple times in different solutions with a new conjunction

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

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

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

function tree_policy($root$, all ASAL args, MC, MBS)
  $leaf =$ None
  With probability p:
    $leaf \leftarrow$ descent to best leaf
    return expand_node($leaf$, all ASAL args, MC, MBS)
  With probability $1-p$:
    return expand_node($root$, all ASAL args, MC, MBS)

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

id:

$$\begin{aligned} &\leftarrow \mathsf{body}(I, J, \_), J \neq 1, \\ &\quad \#\mathsf{count}\{J1 : \mathsf{body}(I, J1, \_), J1 < J\} = 0. \end{aligned} \tag{9}$$

3.5 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 abductive learning process presented in Section 3.4. 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 5). 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 a 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 [10]. 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 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 [10] to select the best child during the *tree policy* phase. All scores (including rewards) are global F_1-scores on the training set.

	Method	Batch F_1-score	MCTS F_1 /*iterations*		$\|States\|$	$\|Guards\|$	Grounding (min)	Solving (min)	Total (min)
			5	10					
(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 6: Experimental results.

3.6 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.

The *activity recognition* data were obtained from the CAVIAR dataset[8], consisting of videos

[7] The code and data are available from https://github.com/nkatzz/asal
[8] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/

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 [44]. 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 6(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 6 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 [50] and EDSM [41], 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 6(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.

3.6.1 Experiments with EVENFLOW Data

Some preliminary experiments regarding automata learning with ASAL on EVENFLOW data are presented in Section 4.

3.7 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 a number of CER datasets, demonstrating empirically their efficacy. Next steps include enhancing the expressive power of the learnt models and further improving scalability.

[9] https://learnlib.de/
[10] This attribute alone is informative enough to learn a useful model in the bio dataset.

4 Use Case Data Exploration and Preliminary Experimental Results

4.1 Personalized Medicine Use Case

The ultimate goal of this use case is forecasting disease progression in breast cancer patients. Unfortunately, we do not have access to a real-world dataset exhibiting disease progression through time, such as deterioration, recovery, or a stable condition. Because of this we turn to synthetic data generation as a means of creating *temporal trajectories* of *virtual patients* [11]. This process begins with genetic information data from breast cancer patients, taken from the TCGA-BRCA dataset [12]. Table 7 below depicts the distribution of the patients chosen from the dataset with respect to the type of breast cancer as well as the cancer stage.

Number of Patients		Cancer Stage			*Total*
		Stage I	Stage II	Stage III	
Cancer Type	Lobular	23	110	36	169
	Ductal	140	446	103	689
Total		163	556	139	858

Table 7: Distribution of breast cancer patients chosen from the TCGA-BRCA dataset

The trajectories ultimately obtained are comprised of sequences of gene expressions "through time" (50 timesteps $\times$ 8954 gene expressions for example). Each trajectory is labelled based on whether this transition signifies stable condition (stage X $\rightarrow$ stage X) or condition deterioration (stage X $\rightarrow$ stage X+1).

4.1.1 Stage Classification

First, we investigate the separability of different stages in a static setting, ignoring the temporal component of the trajectories. As this task is concerned with preliminary exploration, the models used are basic and untuned [13]. All results reported in this section are obtained through a random shuffle of the dataset and a stratified 5-fold cross validation process. The results can be seen in Table 8 below. We observe that while we obtain accuracy scores $> 60\%$, the macro-F1 score, which calculates the metric for each class and finds their unweighted mean, thus overlooking label imbalances, is very poor, indicating that the stages are not separable. What we have confirmed experimentally is a fact well-known in the literature, namely that gene expressions

[11] Creating these is work done solely by Barcelona Supercomputing Center (BSC).
[12] https://portal.gdc.cancer.gov/projects/TCGA-BRCA
[13] In fact, all models shown here have been ran using their vanilla implementation as provided in the respective library (sklearn for all models besides XGBoost).

alone are insufficient for classifying between different cancer stages. This is attributed to very high heterogeneity among patients.

	Classifier	Accuracy	Macro-F1
Lobular	Decision Tree	49.7	**34.5**
	Random Forest	62.7	25.6
	Gradient Boosting	55.7	25.5
	XGBoost	61.6	28.3
	Support Vector Machine	**65.1**	26.3
	Multi-Layer Perceptron	40.4	19.8
Ductal	Decision Tree	49.5	**36.3**
	Random Forest	64.3	28.9
	Gradient Boosting	62.4	31.5
	XGBoost	62.8	32.6
	Support Vector Machine	**64.7**	26.2
	Multi-Layer Perceptron	41.9	22.3

Table 8: Stage classification with gene expression input

Building upon this, we attempt to construct a more compact and informative representation by mapping sets of gene expression to sets of enriched pathways. A biological pathway is a series of actions among molecules in a cell that leads to a certain product or a change in the cell. It can trigger the assembly of new molecules, such as a fat or protein, turn genes on and off, or spur a cell to move [14]. Obtaining pathways from a patient's gene set is achieved through Gene Set Enrichment Analysis (GSEA), a method from the domain of bioinformatics. In most cases this is the product of statistical analysis. Through the use of GSEA, a gene set comprised of several thousands of gene expressions can be mapped to a set of tens of enriched pathways. In essence, the output of GSEA for each pathway is the degree to which this particular biological "function" is overrepresented in this gene set (patient) with regards to a control group.

Table 9 repeats the previous experiment using enriched pathways as the input rather than gene expressions. Since our GSEA tools employ statistical approaches, when the results are statistically inconclusive (i.e. the tool does not have enough information to conclude something) the tool abstains and thus we have no results for that pathway in this case. The dataset in these cases is filled with NaN. In the results below we run one experiment where the NaNs are kept in, in which case we are only able to run classifiers which can deal with this natively, and one experiment where we replace the NaN values with an integer outside the usual range of the features (in this case the feature takes values within $[-4, 4]$ and NaNs are replaced by 20). The motivation behind this choice is twofold: [1] there are now classifiers that can be used that do

[14] https://www.genome.gov/about-genomics/fact-sheets/Biological-Pathways-Fact-Sheet

not deal with NaN values natively, and [2] the classifiers can potentially find correlations and usefulness in the very fact that the value is missing.

We observe that while we are able to obtain marginally better results with several hundred times fewer features, the stages are still unseparable, at least with the basic set of baseline models chosen. Additionally, the NaN imputation appears to be insignificant for the models tested, yielding similar performance for the newly added models, and inducing no difference for the models already being used.

		NaNs kept in		NaNs replaced	
	Classifier	**Accuracy**	**Macro-F1**	**Accuracy**	**Macro-F1**
Lobular	Decision Tree	49.8	**38.5**	49.2	36.1
	Random Forest	**67.5**	35.8	66.9	36.1
	Gradient Boosting	-	-	56.8	32.3
	XGBoost	55.0	32.6	61.6	37.7
	Support Vector Machine	-	-	65.1	26.3
	Multi-Layer Perceptron	-	-	59.8	32.4
Ductal	Decision Tree	49.5	**35.7**	50.2	**35.7**
	Random Forest	63.0	29.1	63.4	29.2
	Gradient Boosting	-	-	58.9	31.4
	XGBoost	58.8	30.3	58.4	30.7
	Support Vector Machine	-	-	**64.7**	26.2
	Multi-Layer Perceptron	-	-	61.0	29.7

Table 9: Stage classification with enriched pathway input

4.1.2 Transition Classification

We move on to transition classification, a task more significant to the use case goal. Since our ultimate aim concerns forecasting if and when a patient will transition to the next stage, we require that, as a minimum, we can classify a transition given the entirety of the sequence. More specifically, using both the start- and end-state, and ignoring all information in between. This data is given in terms of enriched pathways, and so, for each transition, we possess two feature vectors containing patient pathways, one for the start-state and one for the end-state. Table 10 below depicts the distribution over the different constructed transitions.

Provided we have two feature vectors per data point, we perform two experiments. In the first we use the concatenation of the two vectors, creating a new feature vector double the size of the originals, whereas in the second we use the difference between the end- and start-state pathways. The results of these experiments are shown in Table 11 below.

We see that in this setting we are able to classify between different stage transitions much more accurately than we could the individual stages. Furthermore, we observe that using

Number of Transitions		Stage Transition		*Total*
		Stage I → Stage II	Stage II → Stage III	
Cancer Type	Lobular	23	115	138
	Ductal	140	500	640
Total		163	615	778

Table 10: Distribution of stage transitions for breast cancer patients chosen from the TCGA-BRCA dataset

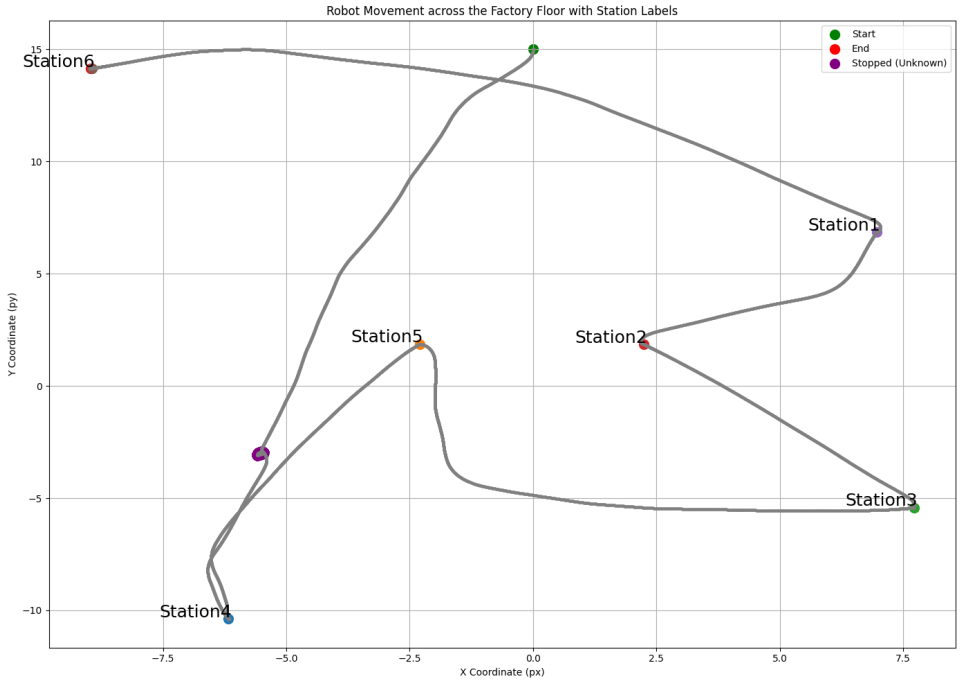


Figure 9: The trace of a robot moving around the smart factory floor.

the difference between the two pathway vectors performs significantly worse than using their concatenation, showing that this is an inferior representation of a transition. Finally, while the NaN imputation appears to lead to marginal performance increases in this case, the metrics reported are similar between the two experiments. The results shown here are promising and encouraging for the downstream task of this use case. That said, as this is preliminary exploration, further analysis is required.

4.2 Industry 4.0 Use Case

The purpose of this use case is to use the EVENFLOW tools in order to learn how to identify and forecast hazardous incidents ahead of time involving autonomous robots in smart factories. The current status of the simulation environment and the generated data is described in deliverable D3.2. We present a brief overview of the data that are currently available. The data consist of

			NaNs kept in		NaNs replaced	
		Classifier	**Accuracy**	**Macro-F1**	**Accuracy**	**Macro-F1**
Concatenation	Lobular	Decision Tree	92.1	85.1	89.9	80.8
		Random Forest	85.6	56.7	86.3	60.2
		Gradient Boosting	-	-	**95.0**	**89.2**
		XGBoost	89.2	76.0	87.7	70.7
		Support Vector Machine	-	-	83.4	45.5
		Multi-Layer Perceptron	-	-	82.7	58.2
	Ductal	Decision Tree	83.8	75.9	85.8	79.0
		Random Forest	89.7	81.8	89.1	80.7
		Gradient Boosting	-	-	89.1	81.5
		XGBoost	90.5	**84.4**	**90.0**	83.4
		Support Vector Machine	-	-	78.0	43.8
		Multi-Layer Perceptron	-	-	84.1	73.9
Difference	Lobular	Decision Tree	74.7	50.8	73.2	53.7
		Random Forest	83.4	49.4	83.4	49.4
		Gradient Boosting	-	-	78.3	50.6
		XGBoost	82.7	51.8	81.9	54.0
		Support Vector Machine	-	-	82.7	45.2
		Multi-Layer Perceptron	-	-	84.1	62.0
	Ductal	Decision Tree	70.6	56.3	72.8	60.1
		Random Forest	78.3	48.9	78.9	52.0
		Gradient Boosting	-	-	80.3	60.3
		XGBoost	78.3	59.7	78.6	59.6
		Support Vector Machine	-	-	78.0	43.8
		Multi-Layer Perceptron	-	-	78.9	61.7

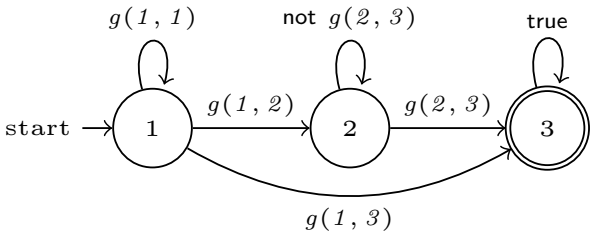
Table 11: Transition classification with enriched pathway input

traces of two robots moving around the smart factory floor, while executing some designated tasks. This involves moving across a number of stopping points (stations), as illustrated in Figure 9. The order in which these stations are to be visited by a robot varies, according to the high-level task that a robot is currently executing.

The data generated by the ongoing simulations consist of robots' mobility data, in addition to images of the surrounding environment captured by the robots' cameras. The former type of time series data may be used for understanding the robots' mobility patterns and learning such patterns to use as the symbolic component in a neuro-symbolic setting, using techniques such as ASAL, presented in Section 3.4. An excerpt of this type of data is presented in Figure 10. Each row in this table corresponds to the values of the available mobility features, namely the

	px	py	pz	ox	oy	oz	ow	vx	vy	vz	action
2	-1.15E-07	14.999998	2.9915218	-0.70703477	6.01E-07	1.76E-06	0.7071788	-3.70E-06	-8.01E-05	1.54E-05	stopped
3	-1.72E-07	14.999997	2.9854658	-0.70703477	9.01E-07	2.64E-06	0.7071788	-3.70E-06	-8.01E-05	1.54E-05	stopped
4	-2.29E-07	14.999996	2.978199	-0.70703477	1.20E-06	3.52E-06	0.7071788	-3.70E-06	-8.01E-05	1.54E-05	moving
5	-2.87E-07	14.999995	2.9697208	-0.70703477	1.50E-06	4.40E-06	0.7071788	-3.70E-06	-8.01E-05	1.54E-05	moving
6	-3.43E-07	14.999994	2.960032	-0.70703477	1.80E-06	5.28E-06	0.7071788	-3.70E-06	-8.01E-05	1.54E-05	moving

Figure 10: An excerpt of time series robot mobility data illustrating the relevant features.



$g(1,1) \leftarrow \text{not } g(1,2), \text{not } g(1,3).$
$g(1,2) \leftarrow \text{at\_least}(o_x, 8), \text{at\_most}(v_y, 2), \text{not } g(1,3).$
$g(2,3) \leftarrow \text{at\_most}(v_y, 3).$
$g(1,3) \leftarrow \text{at\_least}(o_x, 5), \text{at\_least}(w_x, 9), \text{at\_most}(y_y, 6).$

Figure 11: An illustrative symbolic automaton learnt with ASAL from DFKI data.

robot's (x, y, z) coordinates, its quaternion values (ox, oy, oz, ow) and its velocity coordinates (vx, vy, vz) with vz being the rotational velocity across the z-axis. Each row in the dataset is labeled with the robot's immediate destination, i.e. *station 1*, *station 2* and so on. The latter type of data, i.e. images captured by the robots' cameras, may be used as input in the end-goal neuro-symbolic experimental setting, where each robot has to predict the other robot's behavior from such images and a symbolic model of the robot's mobility patterns, given a high-level plan.

We report on preliminary experiments of using ASAL to learn such robot's mobility patterns from the (discretized) time series data, in the form of symbolic automata. A target complex event that is to be captured by such an automaton is the robot's next destination in any point in time, i.e. each automaton captures a mobility pattern towards a different station (*station 1*, *station 2* and so on), which correspond to the classes.

The currently available data sample consists of six trajectories in total, three per robot, for three different high-level execution scenarios/plans. We selected one such trajectory, consisting of 46K time points and segmented it in sequences of length 50, with each trajectory labeled with the class corresponding to the destination of its end point. The segmentation was performed by traversing the trajectory with a sliding window of step 10, resulting in 4595 sub-sequences of length 50. To generate train/test splits we used the first 70% of the sequences for each class for training and retained the remaining 30% for testing. This splitting strategy takes into account the ordering of the sub-sequences in the original trajectory, in order to avoid the degenerated splits resulting from stratified i.i.d sampling, where the testing sequences are overly similar to the training ones.

Each feature signal in the initial multivariate time series trajectory, corresponding to a separate dimension in the time series was discretized beforehand using SAX with 10 bins, similarly to

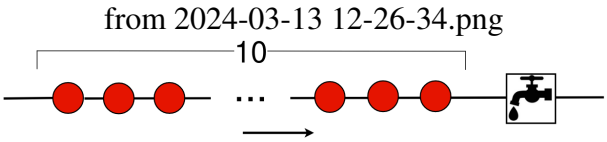
the experiments in Section 3.4. We performed a threefold cross-validation experiment, where ASAL was run with a batch size of 100 for 10 MCTS iterations, a maximum (time limit) of one minute search time per batch and basic feature value comparison predicates as building block predicates. **We obtained an average macro F_1-score of 0.886 and an average running time of 8.3 minutes.** An indicative symbolic automaton learnt in this process is presented in Figure 11. This proof of concept experiment on the currently available, preliminary, yet representative use case data showcases that ASAL can indeed be useful in learning automata-based complex event patterns from EVENFLOW use case data, with the end-goal being to use such patterns in neuro-symbolic settings in the project.

4.3 Infrastructure Lifecycle Assessment Use Case

The goal of this use case is concerned with detecting and pinpointing the location of a leak in a water pipe from accelerometer sensor data.

4.3.1 Experimental Setup and Data

The experimental setup we are concerned with comprises ten accelerometer sensors inside a water pipe, as well as a fire hydrant for simulating a leakage. The arrangement is shown in the diagram of Figure 12a, along with the aerial photo of Figure 12b, which also includes the distances between the sensors and the fire hydrant.



(a) Diagram of experimental setup. Red dots represent sensors and the tap represents a simulated leakage through a fire hydrant.

(b) Aerial photograph of the experimental setup showing the distance between each sensor (marked by white S) and the fire hydrant (red point on the rightmost side).

The data includes ~ 3.5 hours of continuous data collection, during which the fire hydrant was turned on/off. As the original sensor data collection frequency was at $> 1.5kHz$, we sampled the readings at $10Hz$ by averaging all sensor readings within 0.1s, thus obtaining 10 readings per second. This was done to limit the noise of the signal and provide more useful features for the machine learning models. Thus, at the end of the process, for each 0.1s interval we have one

value for each of the ten sensors, as well as a label of whether the hydrant was on or off at that time.

4.3.2 Classification Task

To begin our exploration, we attempt to detect whether the hydrant is on/off based on the readings of all ten sensors. Our models thus accept as input a vector of 10 features, each representing the aggregation of 0.1s worth of that sensor's readings, and outputs a binary label. The label distribution is as follows:

$$\text{fire hydrant on} : 68951 \text{ samples}$$
$$\text{fire hydrant of} : 45071 \text{ samples}$$

Once again, the results reported are the average of the metrics computed on the 5 test sets in the shuffled, 5-fold cross validation. The results of this experiment are shown in Table 12. We see that even with the naive algorithms selected we are able to separate the two classes and can thus detect whether a leak exists in the pipe.

Classifier	Accuracy	Macro-F1
Decision Tree	76.9	75.8
Random Forest	**84.9**	**83.9**
Gradient Boosting	76.1	73.5
XGBoost	83.2	82.2
Support Vector Machine	81.4	79.9
Multi-Layer Perceptron	82.3	81.4

Table 12: Leakage binary classification given input from all ten sensors

4.3.3 Regression Task

We move on to a task closer to the goal of the use case, concerned with pinpointing the location of the leakage. Here we would like our models to accept as input a sequence of readings from a single sensor and determine how far away the leakage is located. Therefore, for this task we construct an alternative dataset which splits up the previous dataset into ten parts, constructing sequences of readings for each sensor individually, and using the distance from that sensor to the leakage point as label. Using Figure 12b as reference we obtain the following:

For this experiment we explore two parameters in generating the dataset. First, the *averaging factor*, which refers to the number of sensor readings that we aggregate into a single number (for

Sensor Number	2	4	6	8	10	3	5	7	9	11
Distance to Leak (m)	7.3	17.4	27.0	35.3	45.3	54.5	65.0	74.8	86.0	93.75

Table 13: Distance from each sensor to the leakage point

example, a factor of 10 aggregates 10 values by taking their mean), and second, the *window size*, which determines the length of the feature vector used. Note that the averaging takes place first, and thus if we have an averaging factor of 2 and window size of 50, we first average every two values and then gather 50 of the resulting numbers as one feature vector. The feature vectors contain readings over a time period of $0.1 \times averaging\_factor \times window\_size$ seconds.

In Table 14 below we show the results of this experiment. The reported metric is the minimum mean average error (MAE) achieved over all models tested. The list of models is identical to that used for previous classification tasks (see Table 12), but in their respective regression forms. In each of the cells we report the MAE and the amount of time "spanned" by one feature vector in that configuration of parameters.

MAE		**Averaging Factor**				
		1	2	5	10	20
Window Size	10	23.4 | $1s$	23.4 | $2s$	23.4 | $5s$	23.3 | $10s$	23.2 | $20s$
	50	23.4 | $5s$	23.3 | $10s$	23.3 | $25s$	23.3 | $50s$	23.4 | $100s$
	100	23.3 | $10s$	23.2 | $20s$	23.3 | $50s$	23.3 | $100s$	23.5 | $200s$
	200	23.2 | $20s$	23.3 | $40s$	23.4 | $100s$	23.6 | $200s$	24.7 | $400s$

Table 14: Leakage distance regression given input readings from one sensor. Each cell includes the minimum mean average error (MAE) achieved over all models tested, as well as the amount of time ”spanned” by one feature vector in that configuration of parameters.

Our main observation is that in this naive form, where we use basic models and do not perform any data processing (such as time series analysis techniques) besides averaging, our regressors are unable to accurately predict the location of the leakage. The second, very surprising, observation is that the length of the feature vector in seconds appears to have minimal effect on the performance of the classifiers. This result requires further investigation.

5 Conclusions and Future Work

This deliverable represents the first step towards neuro-symbolic (NeSy) learning and reasoning for Complex Event Recognition and Forecasting (CER/F) technology in EVENFLOW. We presented the problem setting and the research landscape in terms of the state-of-the art in the field, and we also identified the main challenges involved in combing neural and symbolic models in temporal domains. We tackle some of these challenges by proposing a novel logical/probabilistic framework for complex event pattern specification, which supports scalable probabilistic inference, the translation of such patterns into symbolic automata and their integration with neural predictors in a scalable NeSy training framework tailored specifically for large-scale, temporal applications. This is an important milestone that significantly improves the state-of-the-art, and lays the ground for further progress in NeSy learning and reasoning in the project. We also presented a symbolic learning framework capable of inducing complex event pattern in the form of symbolic automata from multivariate event traces, scales to large datasets via an incremental learning technique and allows to automatically revise such patterns in the face of data drifts. Finally, we presented our work in exploring the available data in EVENFLOW, assessing their quality and drawing insights on how they may be used for the project's end goals. Future work involves:

- Advancements in NeSy learning and reasoning, including differentiable structure learning, as well as jointly training a neural predictor, while learning the structure of the symbolic component in a NeSy architecture.
- Further studying the trade-offs between logical/probabilistic reasoning, which serves as the backbone of differentiable inference, and the expressive power of the utilized symbolic models in EVENFLOW.
- Further scaling-up symbolic learning.
- NeSy complex event forecasting.
- Advancements in forecasts explainability.
- Utilization of data programming techniques towards data augmentation and labeled data generation.
- Thorough evaluation on EVENFLOW's use case data.

5.1 End-to-end Verification of Neuro-Symbolic Systems

An important additional research direction concerns the *verification of temporal neuro-symbolic systems*, which is undergoing work in collaboration with WP5 in EVENFLOW. The main issue that this work addresses is verifying the adversarial robustness of temporal, automata-based

NeSy models in an end-to-end fashion, i.e. going from input perturbations all the way up to the target complex event predictions, in order to assess whether the perceptual input perturbations cause the entire NeSy model to alter its high-level predictions. Note that standard neural network verification techniques are insufficient for such end-to-end verification, since they can only verify "neural-level" properties, expressed as algebraic constraints on the neural network's input-output pairs, with adversarial robustness being a typical example. In a NeSy setting, therefore, such techniques can only be used to verify the neural component, rather than the end-to-end robustness of the entire system w.r.t. its downstream, complex event predictive task. We are working towards end-to-end verification of NeSy systems by combing neural network verification techniques for acquiring simple event verification bounds w.r.t. to the input perturbations, with knowledge compilation and differentiable automata-based inference techniques that allow to compute corresponding bounds on the target complex events via constrained optimization.

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*, pages 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*, pages 427–445. Springer, 2018.

[8] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022.

[9] Suguman Bansal, Yong Li, Lucas Tabajara, and Moshe Vardi. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9766–9774, 2020.

[10] 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.

[11] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

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

[13] 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*, pages 50–61, 2010.

[14] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *International Conference on Computer Aided Verification*, pages 47–67. Springer, 2017.

[15] Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[16] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[17] Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, Rubin Sasha, et al. Pure-past linear temporal and dynamic logic on finite traces. In *IJCAI*, pages 4959–4965, 2020.

[18] Giuseppe De Giacomo and Marco Favorito. Compositional approach to translate ltlf/ldlf into deterministic finite automata. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 122–130, 2021.

[19] Giuseppe De Giacomo, Marco Favorito, Jianwen Li, Moshe Y. Vardi, Shengping Xiao, and Shufang Zhu. Ltlf synthesis as and-or graph search: Knowledge compilation at work. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2591–2598. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[20] Giuseppe De Giacomo, Moshe Y Vardi, et al. Linear temporal logic and linear dynamic logic on finite traces. In *Ijcai*, volume 13, pages 854–860, 2013.

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

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

[23] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

[24] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *International Conference on Extending Database Technology*, pages 627–644. Springer, 2006.

[25] 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, pages 412–422, 2007.

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

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

[28] Francesco Fuggitti. Ltlf2dfa, March 2019.

[29] 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.

[30] 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.

[31] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: a survey. *The VLDB Journal*, 29:313–352, 2020.

[32] 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.

[33] 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.

[34] 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.

[35] Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems: First International Workshop, TACAS'95 Aarhus, Denmark, May 19–20, 1995 Selected Papers 1*, pages 89–110. Springer, 1995.

[36] 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, pages 790–799, 2020.

[37] 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.

[38] Kristian Kersting, Luc De Raedt, and Tapani Raiko. Logical hidden markov models. *Journal of Artificial Intelligence Research*, 25:425–456, 2006.

[39] 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.

[40] 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.

[41] 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*, pages 1–12. Springer, 1998.

[42] 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*, pages 1103–1115, 2021.

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

[44] 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.

[45] Oded Maler and Irini-Eleftheria Mens. A generic algorithm for learning symbolic automata from membership queries. In *Models, Algorithms, Logics and Tools*, pages 146–169. Springer, 2017.

[46] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

[47] 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*, pages 47–58, 2014.

[48] Giuseppe Marra, Sebastijan Dumančić, Robin Manhaeve, and Luc De Raedt. From statistical relational to neurosymbolic artificial intelligence: A survey. *Artificial Intelligence*, page 104062, 2024.

[49] 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.

[50] José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108. World Scientific, 1992.

[51] 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*, pages 62–82. Springer, 2003.

[52] 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.

[53] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. 1995.

[54] 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*, pages 1–12, 2009.

[55] Paulo Shakarian, Chitta Baral, Gerardo I Simari, Bowen Xi, and Lahari Pokala. Neurasp. In *Neuro Symbolic Reasoning and Learning*, pages 63–74. Springer, 2023.

[56] Efthymia Tsamoura, Timothy Hospedales, and Loizos Michael. Neural-symbolic integration: A compositional perspective. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 5051–5060, 2021.

[57] Emile van Krieken, Thiviyan Thanapalasingam, Jakub Tomczak, Frank Van Harmelen, and Annette Ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. *Advances in Neural Information Processing Systems*, 36, 2024.

[58] Jonas Vlasselaer, Wannes Meert, Guy Van den Broeck, and Luc De Raedt. Exploiting local and repeated structure in dynamic bayesian networks. *Artificial Intelligence*, 232:43–53, 2016.

[59] Thomas Winters, Giuseppe Marra, Robin Manhaeve, and Luc De Raedt. Deepstochlog: Neural stochastic logic programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10090–10100, 2022.

[60] 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*, pages 407–418, 2006.

[61] Zhun Yang, Adam Ishay, and Joohyung Lee. Neurasp: Embracing neural networks into answer set programming. *arXiv preprint arXiv:2307.07700*, 2023.

[62] 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*, pages 217–228, 2014.

[63] Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. Symbolic ltlf synthesis. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1362–1369, 2017.