TempAMLSI: Temporal Action Model Learning based on STRIPS translation

Maxence Grand, Damien Pellier, Humbert Fiorino
Univ. Grenoble Alpes, LIG
Grenoble, France
{Maxence.Grand, Damien.Pellier, Humbert.Fiorino}@univ-grenoble-alpes.fr

Abstract

Hand-encoding PDDL domains is generally considered difficult, tedious and error-prone. The difficulty is even greater when temporal domains have to be encoded. Indeed, actions have a duration and their effects are not instantaneous. In this paper, we present TempAMLSI, an algorithm based on the AMLSI approach to learn temporal domains. TempAMLSI is the first approach able to learn temporal domains with Single Hard Envelopes, and TempAMLSI is the first approach able to deal with both partial and noisy observations. We show experimentally that TempAMLSI learns *accurate* temporal domains, i.e., temporal domains that can be used without human proofreading to solve new planning problems with different forms of action concurrency.

1 Introduction

Hand-encoding and proofreading PDDL domains is generally considered difficult, tedious and error-prone by experts, and this is even harder with temporal actions. It is therefore essential to develop tools allowing to automatically learn PDDL domains.

To facilitate non-temporal PDDL domain acquisition, different machine learning algorithms have been proposed: ARMS (Yang, Wu, and Jiang 2007), SLAF (Shahaf and Amir 2006), Louga (Kucera and Barták 2018), LSONIO (Mourão et al. 2012), LOCM (Cresswell, McCluskey, and West 2013), IRale (Rodrigues, Gérard, and Rouveirol 2010), PlanMilner (Segura-Muros, Pérez, and Fernández-Olivares 2018). In these approaches, training data are either (possibly noisy and partial) intermediate states and plans previously generated by a planner, or randomly generated action sequences (i.e. random walks). A major open issue is to learn Temporal PDDL domains (Fox and Long 2003). Temporal PDDL Domains are domains allowing to represent *durative actions*, i.e. actions that have a duration, and whose preconditions and effects must be satisfied and applied at different times. An important property of durative actions is that they can be executed concurrently.

Temporal PDDL domains have different levels of action concurrency (Cushing et al. 2007). Some are sequential, which means that all the plan parts containing overlapping durative actions can be rescheduled into a completely sequential succession of durative actions: each durative action starts after the previous durative action is terminated. One important property of sequential temporal domains is that they can be rewritten as non-temporal domains, and therefore used by classical planners. Some temporal domains require other forms of action concurrency such as Single Hard Envelope (SHE) (Coles et al. 2009). SHE is a form of action concurrency where a durative action can be executed only if another durative action called the envelope extends over it. This is due to the need by the enveloped durative action of a resource, all along its execution, added at the start of the envelope and deleted at the end of the envelope. One important property of SHE temporal domains is that they cannot be sequentially rescheduled. Although some approaches have been proposed to learn temporal features (Gabel and Su 2010; Neider and Gavran 2018; Gaglione et al. 2021; Shah et al. 2018), only (Garrido and Jiménez 2020) proposed an approach learning temporal domains. However this approach is limited to sequential temporal domains. To our best knowledge, there is no learning approach for both SHE and sequential temporal domains.

In this paper, we present TempAMLSI, an accurate learning algorithm for both SHE and sequential temporal domains. TempAMLSI is built on AMLSI (Grand, Fiorino, and Pellier 2020a), an accurate STRIPS domain learner based on grammar induction. Like AMLSI, TempAMLSI takes as input feasible and infeasible action sequences to frame what is allowed by the targeted domain. More precisely, TempAMLSI consists of three steps: (1) TempAMLSI translates temporal sequences into STRIPS sequences, (2) TempAMLSI learns a non-temporal domain with AMLSI, and then (3) translates it into a temporal domain (see Figure - 1).

TempAMLSI contributions in Temporal PDDL domain learning are threefold:

- Temporal actions: TempAMLSI is able to learn both sequential and SHE temporal domains,
- Partial and noisy observations: TempAMLSI is able to learn temporal domains with both partial and noisy observations.
- Accuracy: TempAMLSI is highly accurate even with highly partial and noisy learning datasets: thus, it minimises PDDL proofreading for domain experts. We show

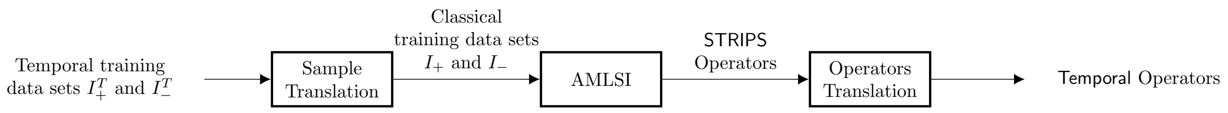
Temporal training data sets I_+^T and I_-^T
Sample Translation
Classical training data sets I_+ and I_-
AMLSI
STRIPS Operators
Operators Translation
Temporal Operators

Figure 1: Overview of the TempAMLSI approach

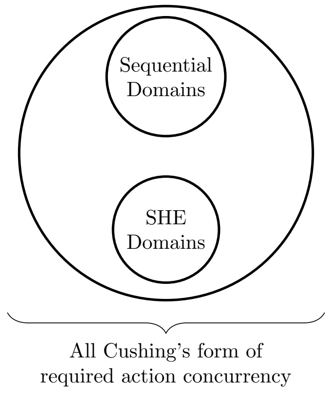

Figure 2: Different forms of required action concurrency

that in many temporal benchmarks AMLSI does not require any correction of the learned domains at all.

The rest of the paper is organized as follows. In section 2 we present a problem statement. In section 3 we give some backgrounds on STRIPS translation and the AMLSI approach, in section 4, we detail TempAMLSI steps. Finally, section 5 evaluates the performance of TempAMLSI on IPC temporal benchmarks.

2 Formal Framework

Section 2.1 introduces a formalization of non-temporal planning domain learning consisting in learning a transition function of a grounded planning domain and in expressing it as PDDL operators. Section 2.2 extends this formalization to temporal domains.

2.1 STRIPS learning problem

A STRIPS planning problem is a tuple $P = (L, A, S, s_0, G, \delta, \lambda)$, where L is a set of logical propositions describing the world states, S is a set of state labels, $s_0 \in S$ is the label of the initial state, and $G \subseteq S$ is the set of goal labels. λ is an observation function $\lambda : S \rightarrow 2^L$ that assigns to each state label the set of logical propositions true in that state. A is a set of action labels. Action preconditions, positive and negative effects are given by the functions $prec$, add and del that are included in $\delta = (prec, add, del)$. $prec$ is defined as $prec : A \rightarrow 2^L$. The functions add and del are defined in the same way. Without loss of generality, we chose this formal framework inspired by (Höller et al. 2016) in order to define the STRIPS learning problem as the lifting of a state transition system into a propositional language.

The function $\tau : S \times A \rightarrow \{true, false\}$ returns whether an action is applicable to a state, i.e. $\tau(s, a) \Leftrightarrow prec(a) \subseteq \lambda(s)$. Whenever action a is applicable in state s_i, the state transition function $\gamma : S \times A \rightarrow S$ returns the resulting state $s_{i+1} = \gamma(s_i, a)$ such that $\lambda(s_{i+1}) = [\lambda(s_i) \setminus del(a)] \cup add(a)$.

A sequence $(a_0 a_1 \ldots a_n)$ of actions is applicable to a state s_0 when each action a_i with $0 \leq i \leq n$ is applicable to the state s_i. Given an applicable sequence $(a_0 a_1 \ldots a_n)$ in state s_0, $\gamma(s_0, (a_0 a_1 \ldots a_n)) = \gamma(\gamma(s_0, a_0), (a_1 \ldots a_n)) = s_{n+1}$. It is important to note that this recursive definition of γ entails the generation of a sequence of states $(s_0 s_1 \ldots s_{n+1})$. A goal state is a state s such that $g \in G$ and $\lambda(g) \subseteq \lambda(s)$. s satisfies g, i.e. $s \models g$ if and only if s is a goal state. An action sequence is a solution plan to a planning problem P if and only if it is applicable to s_0 and entails a goal state.

In formal languages, a set of rules is given that describes the structure of valid words and the language is the set of these words. For STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$, this language is defined as ($0 \leq i \leq n$):

$$\mathcal{L}(P) = \{\omega = (a_0 a_1 \ldots a_n) | a_i \in A, \gamma(s_0, \omega) \models g\}$$

We know that the set of languages generated by STRIPS planning problems are regular languages (Höller et al. 2016). In other words, a STRIPS planning problem $P = (L, A, S, s_0, G, \delta, \lambda)$ generates a language $\mathcal{L}(P)$ that is equivalent to a Deterministic Finite Automaton (DFA) $\Sigma = (S, A, \gamma)$. S and A are respectively the nodes and the edges of the DFA, and γ is the transition function.

For any edge $a \in A$, we call *pre-set* of a the set $\mu_{Ante}(a) = \{s \in S \mid \gamma(s, a) = s'\}$ and *post-set* of a the set $\mu_{Post}(a) = \{s' \in S \mid \gamma(s, a) = s'\}$ (see Figure 3).

A STRIPS learning problem is as follows: given a set of observations $\Omega \subseteq \mathcal{L}(P)$, is it possible to learn the DFA Σ, and then infer P?

For instance, suppose $\Omega = \{a, ab, ba, bab, abb, \ldots\}$ such that $s_0 \xrightarrow{a} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2$, $s_0 \xrightarrow{b} s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_2$, $s_0 \xrightarrow{a} s_2 \xrightarrow{b} s_2 \xrightarrow{b} s_2 \ldots$ (Grand, Fiorino, and Pellier 2020b) show that it is possible to learn Σ (see Figure 3) and infer P with actions $\{a, b\}$, the initial state s_0 and some states marked as goal ($G = \{s_2\}$ in the above example).

2.2 Temporal learning problem

A Temporal planning problem (Fox and Long 2003) is a tuple $TP = (L, A, S, d, s_0, G, \delta, \lambda, T)$. As for STRIPS problems, L is a set of logical propositions, S is a set of state labels, $s_0 \in S$ is the label of the initial state, G is the set of goal labels, and λ is the observation function. A is a set of *durative action* labels. T is an infinite set of timestamps and $d : A \rightarrow \mathbb{R}$ is the duration function. Unlike STRIPS

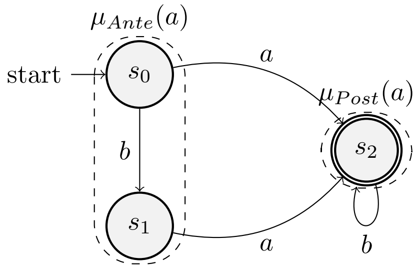

Figure 3: An example of DFA with pre-states and post-states

planning problem, action preconditions, positive and negative effects are labeled with time labels *at-start*, *at-end* and *overall*. More precisely, δ includes:

- $prec : A \times \{s, o, e\} \rightarrow 2^L$: preconditions of $a \in A$ at start, over all, and at end, respectively.
- $add : A \times \{s, e\} \rightarrow 2^L$: positive effects of $a \in A$ at start and at end, respectively.
- $del : A \times \{s, e\} \rightarrow 2^L$: negative effects of $a \in A$ at start and at end, respectively.

The semantics of durative actions is defined in terms of two discrete events $start_a$ and end_a, each of which is naturally expressed as a STRIPS action. Starting a durative action a in state s is equivalent to applying the STRIPS action $start_a$ in s, first verifying that $prec(start_a)$ holds in s. Ending a in state s' is equivalent to applying end_a in s', first by verifying that $prec(end_a)$ holds in s'. $start_a$ and end_a are defined as follows:

$$\begin{array}{lll} start_a: & prec(a,s) = prec(start_a) & add(a,s) = add(start_a) \\ & del(a,s) = del(start_a) & \\ end_a: & prec(a,e) = prec(end_a) & add(a,e) = add(end_a) \\ & del(a,e) = del(end_a) & \end{array}$$

$start_a$ and end_a are constrained by the duration of a, denoted $d(a)$ and the overall precondition: end_a has to occur exactly $d(a)$ time units after $start_a$, and the over all precondition has to hold in all states between $start_a$ and end_a. Although a has a duration, its effects apply instantaneously at the start and the end of a, respectively. The preconditions $prec(a, s)$ and $prec(a, e)$ are also checked instantaneously, but $prec(a, o)$ has to hold for the entire duration of a. The structure of a durative action is summarized in the Figure 4.

A *temporal action sequence* is a set of action-time pairs $\{(a_1, t_1), \dots, (a_n, t_n)\}$. Each action-time pair (a, t) is composed of a durative action $a \in A$ and a scheduled start timestamp $t \in T$ of a, and induces two events $start_a$ and end_a with associated timestamps t and $t + d(a)$, respectively. Events $start_a$ (resp. end_a) is applied in the state s_t (resp. $s_{t+d(a)}$), s_t (resp. $s_{t+d(a)}$) being a state timestamped with t (resp. $t + d(a)$). Then, the temporal transition function γ to learn can be rewritten as: $\gamma(s, a, t) = (\gamma(s_t, start_a), \gamma(s_{t+d(a)}, end_a))$. The transition function $\gamma(s, a, t)$ is defined if and only if: $prec(a, s) \subseteq \lambda(s_t)$, $prec(a, e) \subseteq \lambda(s_{t+d(a)})$ and $\forall t'$ such that $t \leq t' \leq t + d(a)$ $prec(a, o) \subseteq \lambda(s_{t'})$.

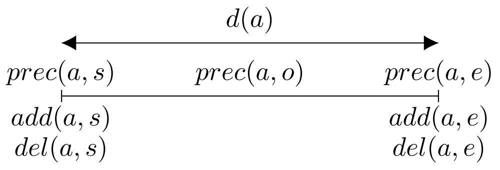

Figure 4: Structure of a durative action a

Finally, we can define a Temporal planning Problem $TP = (L, A, S, d, s_0, G, \delta, \lambda, T)$ as a formal language:

$$\mathcal{L}(TP) = \{\omega = ((a_0, t_0)(a_1, t_1) \dots (a_n, t_n)) | a_i \in A, t_i \in T, g \in G, \gamma(s_0, \omega, t_0) \models g\}$$

A Temporal planning Problem is as follows: given a set of observations $\Omega \subseteq \mathcal{L}(TP)$, is it possible to learn TP?

3 Background

Some planners (Fox and Long 2002; Halsey, Long, and Fox 2004; Celorrio, Jonsson, and Palacios 2015; Furelos Blanco et al. 2018) solve Temporal planning Problems by using non-temporal planners. To that end, they convert Temporal planning Problems into STRIPS planning problems, solve them with a non-temporal planner. Then they convert the classical plan into a temporal plan with rescheduling techniques. In this paper, we build on this idea in order to learn temporal domains: we learn a STRIPS planning problem P and then infer a Temporal planning Problem TP from P.

In this section we present some backgrounds on STRIPS translation techniques. Next, we outline AMLSI on which TempAMLSI is based. AMLSI is an accurate STRIPS learner robust to partial and noisy observations (Grand, Fiorino, and Pellier 2020b,a).

3.1 Temporal planning and STRIPS Translation

Temporal PDDL domains have different levels of required action concurrency (Cushing et al. 2007). Some of them are *sequential*, which means that all the plan parts containing overlapping durative actions can be rescheduled into a completely sequential succession of durative actions: each durative action starts after the previous durative action is terminated. One important property of sequential temporal domains is that they can be rewritten as classical domains, and therefore used by classical non-temporal planners.

To solve a sequential temporal problem, we can translate each durative actions $a \in A$ to a compressed STRIPS action C_a that simulates all of a at once (Coles et al. 2009). The precondition of C_a is the union of the preconditions *at-start* of a with the preconditions *overall* and *at-end* not achieved by the add effect *at-start*. The effect of C_a is the effect *at-start* of a followed immediately by its effect *at-end*. Formally, the compressed action C_a is defined as follows:

- $prec(C_a) = prec(a, s) \cup \{\{prec(a, o) \cup prec(a, e)\} \setminus del(a, s)\}$
- $add(C_a) = \{add(a, s) \setminus del(a, e)\} \cup add(a, e)$
- $del(C_a) = \{del(a, s) \setminus add(a, e)\} \cup del(a, e)$

Once the durative actions are translated, the Temporal planning Problem becomes a STRIPS problem that can be

```
(:durative-action mend
 :parameters(?f-f ?m-m)
 :duration(=?duration 2)
 :condition (and
  (at start(handfree))
  (over all(light ?m)))
 :effect (and
  (at start(not(handfree)))
  (at end(mended ?f))
  (at end(handfree))
  ))
```

(a) Durative declaration of the operator mend

```
(:action mend-start
 :parameters(?f-f ?m-m)
 :precondition (and
  (handfree)
  (light ?m))
 :effect (and
  (not (handfree))))
(:action mend-end
 :parameters(?f-f ?m-m)
 :precondition (and
  (light ?m))
 :effect (and
  (mended ?f)
  (handfree)))
```

(b) STRIPS 2-operators declaration of the operator mend

```
(:action mend-start
 :parameters(?f-f ?m-m)
 :precondition (and
  (handfree))
 :effect (and
  (not (handfree))))
(:action mend-inv
 :parameters(?f-f ?m-m)
 :precondition (and
  (light ?m))
 :effect ())
 (:action mend-end
 :parameters(?f-f ?m-m)
 :precondition ()
 :effect (and (mended ?f)
  (handfree)))
```

(c) STRIPS 3-operators declaration of the operator mend

Figure 5: Comparison between the durative declaration and the classical declaration of the operator "mend" of the Match domain.

solved using a classical planner. When the STRIPS problem is solved, the plan containing compressed actions is translated into a plan with durative actions executed one after another.

Some temporal domains require different forms of action concurrence such as *Single Hard Envelope* (SHE) (Coles et al. 2009). SHE is a form of action concurrency where the execution of a durative action a is required for the execution of a second durative action a'. Formally, a SHE is a durative action a' that adds a proposition p at-start and deletes it at-end while p is an overall precondition of a durative action a. Contrary to sequential temporal domains, for temporal domains containing SHE there exists temporal action sequences that cannot be sequentially rescheduled. For instance, see the Match domain (Figure 5) and the following durative actions:

- $mend(?f\ ?m)$ such that $(light\ ?m) \in prec(mend(?f\ ?m), o)$
- $light(?m)$ such that $(light\ ?m) \in add(light(?m), s)$ and $(light\ ?m) \in del(light(?m), e)$

The durative action $mend(?f\ ?m)$ cannot start before the start of the durative action $light(?m)$ and $mend(?f\ ?m)$ cannot end after the end of $light(?m)$, so $mend(?f\ ?m)$ has to start after the start of $light(?m)$ and to end before the end of $light(?m)$: it is therefore impossible to sequentially reschedule such temporal action sequences.

Generally, to solve SHE Temporal planning problems, planners start by translating durative actions into STRIPS actions. For instance, the CRICKEY planner (Coles et al. 2009) translates each durative action a into three STRIPS actions $start_a, inv_a$ and end_a. Then classical planners are used to solve the problem. Finally, scheduling techniques are used to translate the plans. For instance, the CRICKEY planner builds a set of partially ordered plans with the STRIPS actions. Then, a Simple Temporal Network is used to translate the set of partially ordered plans into a temporal plan.

In addition, it should be noted that there are other forms of required action concurrency besides SHE (Cushing et al. 2007).

3.2 The AMLSI algorithm

AMLSI generates the set of observations Ω by using random walks to learn $\Sigma = (S, A, \gamma)$ and deduce $P = (L, A, S, s_0, G, \delta, \lambda)$. AMLSI assumes L, A, S, s_0 known and the observation function λ possibly partial and noisy (a partial observation is a state where some propositions are missing and a noisy obsevation is a state where the truth value of a proprosition is erroneous). No knowledge of the goal states G is required. Once Σ is learnt, AMLSI has to deduce δ from the transition function γ. Concretely, δ can be represented as a STRIPS planning domain containing all the actions of the problem P and by induction the classical PDDL operators.

The AMLSI algorithm consists of 4 steps: (1) generation of the observations, (2) learning the DFA corresponding to the observations, (3) induction of the PDDL operators from the learnt DFA; (4) finally, refinement of these operators to deal with noisy and partial state observations:

Step 1: AMLSI generates a random walk by applying an action from the initial state of the problem. If the action is applicable in the current state the sequence of actions from the initial state is valid and is added to I^+, the set of positive samples. Otherwise the random walk is stopped and the sequence is added to I_-, the set negative samples.

Step 2: to learn the DFA $\Sigma = (S, A, \gamma)$ AMLSI uses a variant (Grand, Fiorino, and Pellier 2020b) of a classical regular grammar learning algorithm called RPNI (Oncina and García 1992). The learning is based on both I^+ and I^-.

Step 3: AMLSI begins by inducing the preconditions and effects of the actions. For the preconditions $prec(a)$ of action a, AMLSI computes the logical propositions that are in all the states preceding a in Σ:

$$prec(a) = \cap_{s \in \mu_{Ante}(a)} \lambda(s)$$

For the positive effects $add(a)$ of action a, AMLSI computes the logical propositions that are never in states before the execution of a, and always present after a execution:

$$add(a) = \cap_{s \in \mu_{Post}(a)} \lambda(s) \setminus prec(a)$$

Symetrically,

$$del(a) = prec(a) \setminus \cap_{s \in \mu_{Post}(a)} \lambda(s)$$

Once preconditions and effects are induced, actions are lifted to PDDL operators based on OI-subsumption (subsumption under Object Identity) (Esposito et al. 2000): first of all, constant symbols in preconditions and effects are substituted by variable symbols. Then, the less general preconditions and effects, i.e. preconditions and effects encoding as many propositions as possible, are computed as intersection sets. This generalization method allows to ensure that all the necessary preconditions, i.e. the preconditions allowing to differentiate the states where actions are applicable from states where they are not, to be rightfully coded in the corresponding operators.

Step 4: to deal with noisy and partial state observations, AMLSI starts by refining the operator effects to ensure that the generated operators allow to regenerate the induced DFA. To that end, AMLSI adds all the effects ensuring that each transition in the DFA are feasible. Then, AMLSI refines the preconditions of the operators. As in (Yang, Wu, and Jiang 2007), it makes the following assumptions: the negative effects of an operator must be used in its preconditions. Thus, for each negative effect of an operator, AMLSI adds the corresponding propositions in the preconditions. Since effect refinements depend on preconditions and precondition refinements depend on effects, AMLSI repeats these two refinements steps until convergence, i.e., no more precondition or effect is added. Finally, AMLSI performs a Tabu Search to improve the PDDL operators independently of the induced DFA, on which operator generation is based. Once the Tabu Search reaches a local optimum, AMLSI repeats all the three refinement steps until convergence.

4 Temporal AMLSI

TempAMLSI (summarized in Figure - 1) is an extension of AMLSI. It has three steps. (1) After having generated the samples of *temporal* sequences (including both feasible and infeasible sequences), TempAMLSI translates them into non-temporal sequences (see Section - 4.1), (2) TempAMLSI uses AMLSI to learn STRIPS operators (see Section - 3), and (3) translates them into Temporal operators (see Section - 4.2).

In practice, the temporal sequences generated by TempAMLSI are timestamped start and end event sequences. For instance:

$$\{(0, start\ light(m)), (0.5, start\ mend(f_1, m)), (2.5, end\ mend(f_2, m)), (2.6, start\ mend(f_2, m)), (4.6, end\ mend(f_2, m)), (5, end\ light(m))\}$$

Meaning durative action $light(m)$ starts at 0 and finishes at 5, $mend(f_1, m)$ starts at 0.5 and finishes at 2.5 and $mend(f_2, m)$ starts at 2.6 and finishes at 4.6. In the rest of this section we focus on the sample and operator translation steps. We will present two variants for these translations:

2-Operators Translation: The STRIPS action sequences contain, for each durative action a, the start action $start_a$ and the end action end_a corresponding to the events of a durative action (see Section - 2.2). This method only translates the events observed in the temporal sequences. But, it does not directly represent the *overall* preconditions that constrain the "life cycle" of a durative action. Indeed, for a durative action a to be executed the *at-start* preconditions must be checked at the start event, and the *at-end* preconditions must be checked at the end event, but it is also necessary that the *overall* preconditions are satisfied on all the duration of action a.

3-Operators Translation: The STRIPS action sequences contain, for each durative action a, as for the 2-Operators translation, the start action $start_a$ and the end action end_a corresponding to the events of a durative action. However, they also contains inv_a: an invariant action. This invariant action allows to represent the *overall* preconditions.

4.1 Sample Translation

Let ω^T be a temporal sequence such that:

$$\omega^T = \{(0, start\ light(m)), (0.5, start\ mend(f_1, m)), (2.5, end\ mend(f_1, m)), (2.6, start\ mend(f_2, m)), (4.6, end\ mend(f_2, m)), (5, end\ light(m))\}$$

2-Operators Each durative action a is converted into two event actions $start_a$ and end_a. After conversion:

$$\begin{aligned}\omega = \ & \{start_{light(m)}, start_{mend(f_1,m)}, end_{mend(f_1,m)},\\ & start_{mend(f_2,m)}, end_{mend(f_2,m)}, end_{light(m)}\}\end{aligned}$$

3-Operators In this case, each durative action a is converted into three event actions $start_a$ inv_a and end_a. After conversion, we have the following sequence:

$$\begin{aligned}\omega = \ & \{start_{light(m)}, inv_{light(m)}, start_{mend(f_1,m)},\\ & inv_{light(m)}, inv_{mend(f_1,m)}, end_{mend(f_1,m)},\\ & inv_{light(m)}, start_{mend(f_2,m)}, inv_{light(m)},\\ & inv_{mend(f_2,m)}, end_{mend(f_2,m)}, end_{light(m)}\}\end{aligned}$$

4.2 Operators Translation

After having learnt the STRIPS domain with AMLSI, TempAMLSI converts STRIPS operators into temporal operators. The Figure - 5 gives an example of conversion for the *mend* operator of the Match domain for the 2-Operators and 3-Operators variants.

2-Operators translation is as follows:

$$\begin{aligned}
&prec(a, s) = prec(start_a) \setminus prec(end_a)\\
&add(a, s) = add(start_a)\\
&del(a, s) = del(start_a)\\
&prec(a, e) = prec(end_a) \setminus prec(start_a)\\
&add(a, e) = add(end_a)\\
&del(a, e) = del(end_a)\\
&prec(a, o) = prec(start_a) \cap prec(end_a)
\end{aligned}$$

at-start (resp. *at-end*) effects are the effects of start (resp. end) STRIPS operators. *overall* preconditions are the intersection of preconditions of start and end STRIPS operators. And, *at-start* (resp. *at-end*) preconditions are the preconditions of the start (resp. end) STRIPS operators excluding end (resp. start) preconditions.

3-Operators translation is as follows:

$$\begin{aligned}
&prec(a, s) = prec(start_a)\\
&add(a, s) = add(start_a)\\
&del(a, s) = del(start_a)\\
&prec(a, e) = prec(end_a))\\
&add(a, e) = add(end_a)\\
&del(a, e) = del(end_a)\\
&prec(a, o) = prec(inv_a)
\end{aligned}$$

at-start (resp. *at-end*) effects are the effects of start (resp. end) STRIPS operators. *overall* preconditions are the preconditions of *inv* STRIPS operators. And, *at-start* (resp. *at-end*) preconditions are the preconditions of start (resp. end) STRIPS operators.

5 Experiments and evaluations

The evaluation consists in the comparison of the performance of the 2-Operators and 3-Operators variants (see Section - 4) of TempAMLSI. First, we compare the performance of both variants for a given learning dataset. In order to study the performance of both variants with respect to noisy and partial state observations, we use six different experimental scenarios classically used to evaluate domain learning algorithm:

1. Complete intermediate observations (100%) and no noise (0%)
2. Complete intermediate observations (100%) and low level of noise (1%)
3. Complete intermediate observations (100%) and high level of noise (10%)
4. Partial intermediate observations (25%) and no noise (0%)
5. Partial intermediate observations (25%) and low level of noise (1%)
6. Partial intermediate observations (25%) and high level of noise (10%)

Then, we compare the performance of both variants as a function of the size of the training dataset.

5.1 Experimental setup

Our experiments are based on 5 temporal IPC domains[1] (see Table - 1). More precisely we test TempAMLSI with three Sequential domains (Peg Solitaire, Sokoban, Zenotravel), and two SHE domains (Match, Turn and Open). We test each IPC domain with 3 different initial states over five runs, and we use five seeds randomly generated for each run. Then, the length of the random walk sequences is randomly chosen between 10 and 30 durative actions. Finally we generate partial observations by randomly removing a fraction of the propositions of the states, and we generate noise by changing the value of a fraction of the observable propositions. All the tests were performed on an Ubuntu 14.04 server with a multi-core Intel Xeon CPU E5-2630 clocked at 2.30 GHz with 16GB of memory. PDDL4J library (Pellier and Fiorino 2018) was used to generate the benchmark data.[2]

5.2 Evaluation Metrics

We evaluate TempAMLSI with three different metrics: the *syntactical error* (Zhuo et al. 2010) that computes the distance between the original domain and the domain learnt, the *accuracy* (Zhuo, Nguyen, and Kambhampati 2013) that expresses the capability of the domain learnt to solve new problems (without proofreading). Even though the syntactical error is the most used metric in the literature, we argue that the accuracy is the most important metric *in pratice* for planning because it measures to what extent a learnt domain is useful. Indeed, it often happens that one missing precondition or effect, which amounts to a small syntactical error, makes the learnt domain unable to solve planning problems. Finally, the last metric is the *FScore* that expresses the capability of the learnt domain to satisfy the level of required action concurrency in the original domain.

Formally, the syntactical error $error(a)$ for an action a is defined as the number of extra or missing predicates in the preconditions $prec(a)$, the positive effects $add(a)$ and the negative effects $del(a)$ divided by the total number of possible predicates (Hamming distance). By extension, the syntactical error for a domain composed of a set of actions A is: $E_\sigma = \frac{1}{|A|}\sum_{a\in A} error(a)$.

$Accuracy = \frac{N}{N^*}$ is the ratio between N, the number of correctly solved problems with the learnt domain, and N^*, the total number of problems to solve. In the rest of this section the accuracy is computed over 20 problems. We also report in our results the ratio of (possibly incorrectly) solved problems. A problem is incorrectly solved when a solution plan is found with the learnt domain that is not correct with respect to the original domain. In the experiments, we solve the benchmark problems with the TP-SHE (Celorrio, Jonsson, and Palacios 2015) planner. Plan validation is done with VAL, the IPC competition validation tool (Howey and Long 2003).

Finally, FScore $= \frac{2.P.R}{P+R}$ where R is the recall, i.e. the rate of sequences e accepted by the original IPC domain that are successfully accepted by the learnt domain, computed as $R = \frac{|\{e\in E^+ \mid accept(\delta,e)\}|}{|E^+|}$, and P is the precision, i.e. the rate of sequences e accepted by the learnt domain that are also accepted by the original IPC domain, computed as $P = \frac{|\{e\in E^+ \mid accept(\delta,e)\}|}{|\{e\in E^+ \mid accept(\delta,e)\}\cup\{e\in E^- \mid accept(\delta,e)\}|}$. The test sets E^+ and E^- used to compute the FScore are generated by random walks as in Section - 3.

[1]https://www.icaps-conference.org/competitions/

[2]The experimental setup and results are uploaded at https://gitlab.com/grandmaxIm2ag/amlsi/.

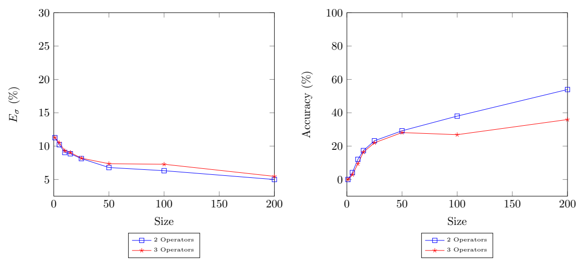

Figure 6: Performance of 2-Operators and 3-Operators variants when the training data set size increases in number of actions

5.3 Results

Sequential Temporal Domains Table - 2 gives the results for Sequential Temporal Domains. For Sequential Temporal Domains, we can observe that the 2-Operators variant is gen-

Domain	Operators	Predicates	Type	$\lvert I_+\rvert$	$\lvert I_-\rvert$	$\lvert E_+\rvert$	$\lvert E_-\rvert$	$\lvert\omega_+^T\rvert$	$\lvert\omega_-^T\rvert$	$\lvert e_+\rvert$	$\lvert e_-\rvert$
Peg Solitaire	1	3	Sequential	30	1293	100	4314	4	4	4	4
Sokoban	2	3	Sequential	30	13510	100	45000	20	12	20	12
Zenotravel	5	4	Sequential	30	5555	100	18273	20	12	20	12
Match	2	4	SHE	30	983	100	3308	4	4	4	4
Turn and Open	5	8	SHE	30	5635	100	18523	20	11	20	11

Table 1: Benchmark domain characteristics. The columns indicate respectively the number of operators and predicates, the temporal domain type, the number of sequences in I_+, I_-, E_+ and E_-. E_+ and E_- are the test sets for the FScore, the average length of the positive (resp. negative) training sequences $\omega_+^T \in I_+$ (resp. $\omega_-^T \in I_-$) and the average length of the positive (resp. negative) test sequences $e_+ \in E_+$ (resp. $e_- \in E_-$)

Domain	Observability	Noise	Algorithm	E_σ	FScore	Solved	Accuracy	Time
Peg	100%	0%	2-Operators	0.9	100	100	100	14
			3-Operators	0.9	100	100	100	36.9
		1%	2-Operators	0.9	100	100	100	39.7
			3-Operators	0.9	100	100	100	19.6
		10%	2-Operators	**0.9**	100	100	100	31.9
			3-Operators	1.3	100	100	100	41.1
	25%	0%	2-Operators	0.9	100	100	100	34.1
			3-Operators	0.9	100	100	100	77.3
		1%	2-Operators	1	100	100	100	83.7
			3-Operators	1	100	100	100	49.6
		10%	2-Operators	**1.4**	**98.7**	**100**	**100**	73.2
			3-Operators	3.5	86.6	86.7	62.7	84.7
Zenotravel	100%	0%	2-Operators	2.4	100	100	100	370.3
			3-Operators	2.4	100	100	100	483.6
		1%	2-Operators	**2.7**	**92.6**	**93.3**	**93.3**	665.3
			3-Operators	3.4	58.3	83.3	34	447
		10%	2-Operators	**3.4**	**81.8**	**97.7**	**80**	554.1
			3-Operators	3.8	26.6	87	55	495
	25%	0%	2-Operators	**2.4**	**100**	**100**	**100**	923
			3-Operators	2.9	86.3	90.7	73.3	1279.8
		1%	2-Operators	**3.5**	**71.2**	**93.3**	**73.3**	1417
			3-Operators	4.1	51.7	90.3	33.3	985.7
		10%	2-Operators	5.6	**31.5**	69.7	**56.3**	1346.3
			3-Operators	**5.3**	26.9	**83.7**	37	1921.3
Sokoban	100%	0%	2-Operators	0	100	100	100	289.5
			3-Operators	0	100	100	100	582.9
		1%	2-Operators	**1.9**	**58.6**	100	**86.7**	476
			3-Operators	2.3	27.7	100	80	1150.9
		10%	2-Operators	**2.9**	**77.7**	**78.7**	**58.7**	609.3
			3-Operators	4.2	16.6	86.7	46.7	1364.9
	25%	0	2-Operators	0	100	100	100	315.3
			3-Operators	0	100	100	100	550.4
		1%	2-Operators	**2**	**57.6**	100	100	469.7
			3-Operators	2.8	38.8	100	100	1132.1
		10%	2-Operators	**3.2**	**61.3**	**86.7**	**86.7**	487.9
			3-Operators	5.3	29.3	64.7	29.3	1248.7

Table 2: Domain learning results on sequential temporal domains. For each metric, results are in percentage and the best results are in bold. Learning time is in seconds.

erally more robust than the 3-Operators variant. Also, in the majority of domains TempAMLSI learns accurate domains ($Accuracy \geq 50\%$) when observations are complete whatever the level of noise. When observations are partial, TempAMLSI is generally not able to learn accurate domains with a high level of noise. Also, we can note that in 2 domains (Peg-Solitaire and Zenotravel) the syntactical error is not equal to 0 when accuracy is optimal. This is because TempAMLSI learns implicit preconditions that are not encoded in the IPC domain. An implicit precondition is a precon-

Domain	Observability	Noise	Algorithm	E_σ	FScore	Solved	Accuracy	Time
Match	100%	0%	2-Operators	3.6	97.9	100	100	7.2
			3-Operators	3.6	87.6	100	100	13.1
		1%	2-Operators	4	**89.3**	93.3	93.3	16.2
			3-Operators	**3.7**	84.2	**100**	**100**	15.7
		10%	2-Operators	**4**	**83.7**	80	80	16.2
			3-Operators	4.5	77.9	**100**	**100**	38
	25%	0%	2-Operators	**4.8**	**83.6**	**100**	**100**	20.4
			3-Operators	6.1	56.7	93.3	93.3	31.7
		1%	2-Operators	**4.8**	**87.7**	93.3	93.3	38.3
			3-Operators	5.6	58.9	93.3	93.3	27.2
		10%	2-Operators	**6.7**	**84.6**	**86.7**	**86.7**	27.4
			3-Operators	8.1	49.3	66.7	66.7	31.2
Turn and Open	100%	0%	2-Operators	**1.2**	**100**	95	**95**	295.7
			3-Operators	1.5	84.3	95	85	904.9
		1%	2-Operators	**1.9**	**86**	88.3	**79**	1043
			3-Operators	2.6	72.5	**89.3**	64.7	841.1
		10%	2-Operators	**4**	**60.8**	74.3	**23.7**	759
			3-Operators	4.2	58.4	**75**	18.3	973.8
	25%	0%	2-Operators	2.4	**98**	56.7	50	1259.1
			3-Operators	2.4	77.2	**92.3**	**63.7**	223.4
		1%	2-Operators	3.5	82	44.3	37	2811.1
			3-Operators	**2.3**	**77**	**88.7**	**70.3**	2478.2
		10%	2-Operators	**5.6**	**44.7**	63	**28.3**	2582.8
			3-Operators	5.7	39	**87.3**	23	2943.2

Table 3: Domain learning results on SHE temporal domains. For each metric, results are given in percentage and the best results are in bold. Learning time is in seconds.

dition that is implied by another precondition. Also, some *at-start* preconditions are encoded as *overall* preconditions in the learnt domains, and vice versa.

SHE Temporal Domains Table - 3 gives the results for SHE Temporal Domains. For SHE Temporal Domains, we can observe that the 2-Operators variant is generally more robust than the 3-Operators variant when observations are complete whatever the level of noise. When observations are partial the 2-Operators variant is more robust than the 3-Operators variant only for the Match domains. Also, for all domains TempAMLSI learns accurate domains when the level of noise is not high whatever the level of observability. With a high level of noise TempAMLSI learns accurate domains only for the Match domains.

The fact that the 2-Operators variant is more robust than the 3-Operators variant can be explained in different ways. First of all, the fact that action sequences of 2-Operators variants are shorter than action sequences of 3-Operators variants makes DFAs easier to learn since they have fewer states. The better the DFA learning, the better the operator learning. Moreover, it is easier for 2-Operators variants than for 3-Operators variants because 2-Operators variants have less operators. Finally, Figure 6 shows the average performance of 2-Operators and 3-Operators obtained on the 5 domains of our benchmarks when varying the training data set size. The size of the training set is indicated in number of actions. For the sake of compactness, we present here only the results obtained on the most difficult scenario 6 (partial intermediate observations (25%) and high level of noise (10%)). We observe that the 2-Operators variant needs very little data to obtain a relatively large accuracy (almost 50% with only a learning dataset of 200 actions) in the most difficult scenario. Also, we observe that the 2-Operators variant outperforms the 3-Operators variant with little data.

6 Conclusion

In this paper we have presented TempAMLSI, a novel algorithm to learn temporal PDDL domains. TempAMLSI is built on the AMLSI approach and the idea to use classical PDDL domain translation: TempAMLSI converts temporal action sequences into non-temporal sequences. Then TempAMLSI uses AMLSI algorithm to learn a classical PDDL domain and converts it into a temporal PDDL domain. Our experimental results show that TempAMLSI is able to learn accurately both sequential and SHE temporal domains from partial and noisy datasets. However, SHE are not the only form of required action concurrency. Indeed, there exist different levels of required action concurrency for each Allen's interval algebra. So in future works, TempAMLSI will be extended to encompass more temporal relations.

Acknowledgments

This research is supported by the French National Research Agency under the "Investissements d'avenir" program (ANR-15-IDEX-02) on behalf of the Cross Disciplinary Program CIRCULAR.

References

Celorrio, S. J.; Jonsson, A.; and Palacios, H. 2015. Temporal Planning With Required Concurrency Using Classical Planning. In *Proc. of ICAPS*, 129–137.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1): 1–44.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using *LOCM*. *Knowledge Engineering Review*, 28(2): 195–213.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is Temporal Planning Really Temporal? In *Proc. of IJCAI*, 1852–1859.

Esposito, F.; Semeraro, G.; Fanizzi, N.; and Ferilli, S. 2000. Multistrategy Theory Revision: Induction and Abduction in INTHELEX. *Machine Learning*, 38(1-2): 133–156.

Fox, M.; and Long, D. 2002. Fast Temporal Planning in a Graphplan Framework. In *Proc. of AIPS workshop*, 9–17.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Furelos Blanco, D.; Jonsson, A.; Palacios Verdes, H. L.; and Jiménez, S. 2018. Forward-search temporal planning with simultaneous events. In *Proc. of Constraint Satisfaction Techniques for Planning and Scheduling Workshop*.

Gabel, M.; and Su, Z. 2010. Online inference and enforcement of temporal properties. In *Proc. of ICSE*, 15–24.

Gaglione, J.; Neider, D.; Roy, R.; Topcu, U.; and Xu, Z. 2021. Learning Linear Temporal Properties from Noisy Data: A MaxSAT-Based Approach. In *Proc. of ATVA*, 74–90.

Garrido, A.; and Jiménez, S. 2020. Learning Temporal Action Models via Constraint Programming. In *Proc. of ECAI*, 2362–2369.

Grand, M.; Fiorino, H.; and Pellier, D. 2020a. AMLSI: A Novel and Accurate Action Model Learning Algorithm. In *Proc. of the International Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Grand, M.; Fiorino, H.; and Pellier, D. 2020b. Retro-engineering state machines into PDDL domains. In *Proc. of ICTAI)*, 1186–1193.

Halsey, K.; Long, D.; and Fox, M. 2004. CRIKEY-a temporal planner looking at the integration of scheduling and planning. In *Proc. of the Integrating Planning into Scheduling Workshop*, 46–52.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the Expressivity of Planning Formalisms through the Comparison to Formal Languages. In *Proc. of ICAPS*, 158–165.

Howey, R.; and Long, D. 2003. VAL's progress: The automatic validation tool for PDDL2. 1 used in the international planning competition. In *Proc. of the International Workshop on the International Planning Competition*, 28–37.

Kucera, J.; and Barták, R. 2018. LOUGA: Learning Planning Operators Using Genetic Algorithms. In *Proc. of PKAW Workshop*, 124–138.

Mourão, K.; Zettlemoyer, L. S.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Proc. of UAI*, 614–623.

Neider, D.; and Gavran, I. 2018. Learning Linear Temporal Properties. In *Proc. of FMCAD*, 1–10.

Oncina, J.; and García, P. 1992. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis: Selected Papers from the IVth Spanish Symposium*, volume 1, 49–61. World Scientific.

Pellier, D.; and Fiorino, H. 2018. PDDL4J: a planning domain description library for java. *Journal of Experimental and Theoretical Artificial Intelligence*, 30(1): 143–176.

Rodrigues, C.; Gérard, P.; and Rouveirol, C. 2010. Incremental Learning of Relational Action Models in Noisy Environments. In *Proc. of ILP*, 206–213.

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2018. Learning Numerical Action Models from Noisy and Partially Observable States by means of Inductive Rule Learning Techniques. In *Proc. of KEPS workshop*.

Shah, A.; Kamath, P.; Shah, J. A.; and Li, S. 2018. Bayesian Inference of Temporal Task Specifications from Demonstrations. In *Prof. of NeurIPS*, 3808–3817.

Shahaf, D.; and Amir, E. 2006. Learning Partially Observable Action Schemas. In *Proc. of AAAI Conference on Artificial Intelligence*, 913–919.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3): 107–143.

Zhuo, H.; Yang, Q.; Hu, D.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18): 1540–1569.

Zhuo, H. H.; Nguyen, T. A.; and Kambhampati, S. 2013. Refining Incomplete Planning Domain Models Through Plan Traces. In *Proc. of IJCAI*, 2451–2458.