FOND Planning for LTL$_f$ and PLTL$_f$ Goals

Extract of MSc. Thesis

Francesco Fuggitti

Disclaimer for the Reader

This is an extract of my Master thesis titled "LTL and Past LTL on Finite Traces for Planning and Declarative Process Mining". The thesis was carried out in 2018 at Sapienza University under the supervision of Prof. Giuseppe De Giacomo. Since then, some tools developed in the thesis have been updated to major releases. Therefore, some of statements present in this extract may be referring to older releases of such tools.

1 Introduction

In this section, we will define a new approach to the problem of non-deterministic planning for extended temporal goals. In particular, we will give a solution to this problem reducing it to a *fully observable non deterministic* (FOND) planning problem and taking advantage of our tool LTL$_f$2DFA[1]. First of all, we will introduce the main idea and motivations supporting our approach. Then, we will give some preliminaries explaining the Planning Domain Definition Language (PDDL) language and the FOND planning problem formally. After that, we will illustrate our FOND4LTL$_f$/PLTL$_f$ (available online at http://fond4ltlfpltl.diag.uniroma1.it/) approach with the encoding of temporal goals into a PDDL domain and problem. Finally, we will present our practical implementation of the proposed solution.

[1] http://ltlf2dfa.diag.uniroma1.it/

2 Idea and Motivations

Planning for temporally extended goals with *deterministic* actions has been well studied during the years starting from (Bacchus and Kabanza, 1998) and (Doherty and Kvarnstram, 2001). Two main reasons why temporally extended goals have been considered over the classical goals, viewed as a desirable set of final states to be reached, are because they are not limited in what they can specify and they allows us to restrict the manner used by the plan to reach the goals. Indeed, temporal extended goals are fundamental for the specification of a collection of real-world planning problems. Yet, many of these real-world planning problems have a *non-deterministic* behavior owing to unpredictable environmental conditions. However, planning for temporally extended goals with *non-deterministic* actions is a more challenging problem and has been of increasingly interest only in recent years with (Camacho et al., 2017; De Giacomo and Rubin, 2018).

In this scenario, we have devised a solution to this problem that exploits the translation of a temporal formula to a DFA, using LTL$_f$2DFA. In particular, our idea is the following: given a non-deterministic planning problem and a temporal formula, we first obtain the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encode such a DFA into the non-deterministic planning domain. As a result, we have reduced the original problem to a classic FOND planning problem. In other words, we compile extended temporal goals together with the original planning domain, specified in PDDL, which is suitable for input to standard (FOND) planners.

3 Preliminaries

In this section, we will give some basics on the PDDL specification language for domains and problems of planning and a general formalization of FOND planning.

3.1 PDDL

As stated before, PDDL is the acronym for Planning Domain Definition Language, which is the *de-facto* standard language for representing "classical" planning tasks. A general planning task has the following components:

- Objects: elements in the world that are of our interest;
- Predicates: objects properties that can be true or false;

- Initial state: state of the world where we start;
- Goal state: things we want to be true;
- Action/Operator: rule that changes the state.

Moreover, planning tasks are composed by two files: the *domain* file where are defined predicates and actions and a *problem* file where are defined objects, the initial state and the goal specification.

3.1.1 The *domain* file

The *domain* definition gives each domain a name and specifies predicates and actions available in the domain. It might also specify types, constants and other things. A simple domain has the following format:

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl] ...)
  [(:types T1 T2 T3 T4 ...)]
  (:predicates (PREDICATE_1_NAME [?A1 ?A2 ... ?AN])
               (PREDICATE_2_NAME [?A1 ?A2 ... ?AN])
              ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
   )
  (:action ACTION_2_NAME
    ...)
  ...)
```

where `[]` indicates optional elements. To begin with, any PDDL *domain* definition must declare its expressivity requirements given after the `:requirements` key. The basic PDDL expressivity is called STRIPS[2], whereas a more complex one is the Action Description Language (ADL), that extends STRIPS in several ways, such as providing support for negative preconditions, disjunctive preconditions, quantifiers, conditional effects etc.. Nevertheless, many planners do not support full ADL because creating plans efficiently is not trivial. Although the presence of this limitation, the PDDL language allows us to use

[2] STRIPS stands for STanford Research Institute Problem Solver, which is a formal language of inputs to the homonym automated planner developed in 1971.

only some of the ADL features. Furthermore, there are also other requirements often used that can be specified as `equality`, allowing the usage of the predicate `=` interpreted as equality, and `typing` allowing the typing of objects. As we will explain later, our practical implementation supports, so far, only simple ADL, namely conditional effects in domain's operators which do not have any nested subformula.

Secondly, there is the predicates definition after the `:predicates` key. Predicates may have zero or more parameters variables and they specify only the number of arguments that a predicate should have. Moreover, a predicate may also have typed parameters written as `?X -- TYPE_OF_X`.

Thirdly, there is a list of action definitions. An action is composed by the following items:

- *parameters*: they stand for free variables and are represented with a preceding question mark `?`;
- *precondition*: it tells when an action can be applied and, depending on given requirements, it could be differently defined (i.e. conjunctive formula, disjunctive formula, quantified formula, etc.);
- *effect*: it tells what changes in the state after having applied the action. As for the precondition, depending on given requirements, it could be differently defined (i.e. conjunctive formula, conditional formula, universally quantified formula, etc.)

In particular, in pure STRIPS domains, the precondition formula can be one of the following:

- an atomic formula as `(PREDICATE_NAME ARG1 ... ARG_N)`
- a conjunction of atomic formulas as `(and ATOM1 ... ATOM_N)`

where arguments must either be parameters of the action or constants.

If the *domain* uses the `:adl` or `:negated-precondition` an atomic formula could be expressed also as `(not (PREDICATE_NAME ARG1 ... ARG_N))`. In addition, if the domain uses `:equality`, an atomic formula may also be of the form `(= ARG1 ARG2)`.

On the contrary, in ADL domains, a precondition formula could be one of the following:

- a general negation as `(not CONDITION_FORMULA)`
- a conjunction of condition formulas as `(and CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`

- a disjunction of condition formulas as `(or CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- an implication as `(imply CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- an implication as `(imply CONDITION_FORMULA1 ... CONDITION_FORMULA_N)`
- a universally quantified formula as `(forall (?V1 ?V2 ...) CONDITION_FORMULA)`
- an existentially quantified formula as `(exists (?V1 ?V2 ...) CONDITION_FORMULA)`

The same division can be carried out with effects formulas. Specifically, in pure STRIPS domains, the precondition formula can be one of the following:

- an added atom as `(PREDICATE_NAME ARG1 ... ARG_N)`
- a deleted atom as `(not (PREDICATE_NAME ARG1 ... ARG_N))`
- a conjunction of effects as `(and ATOM1 ... ATOM_N)`

On the other hand, in an ADL domains, an effect formula can be expressed as:

- a conditional effect as `(when CONDITION_FORMULA EFFECT_FORMULA)`, where the `EFFECT_FORMULA` is occur only if the `CONDITION_FORMULA` holds true. A conditional effect can be placed within quantification formulas.
- a universally quantified formula as `(forall (?V1 ?V2 ...) EFFECT_FORMULA)`

As last remark that we will deepen later in Section 3.2, when the PDDL *domain* has *non-deterministic* actions, the effect formula of those actions expresses the non-determinism with the keyword `oneof` as `(oneof (EFFECT_FORMULA_1) ... (EFFECT_FORMULA_N)`.

In the following, we show a simple example of PDDL *domain*.

Example 3.1. A simple PDDL *domain* the Tower of Hanoi game. This game consists of three rods and n disks of different size, which can slide into any rods. At the beginning, disks are arranged in a neat stack in ascending order of size on a rod, the smallest on the top. The goal of the game is to move the whole stack to another rod, following three rules:

- one disk at a time can be moved;
- a disk can be moved only if it is the uppermost disk on a stack;
- no disk can be placed on top of a smaller disk.

```
1  (define (domain hanoi) ;comment
2    (:requirements :strips :negative-preconditions :equality)
3    (:predicates (clear ?x) (on ?x ?y) (smaller ?x ?y) )
4    (:action move
5      :parameters (?disc ?from ?to)
6      :precondition (and
7          (smaller ?disc ?to) (smaller ?disc ?from)
8          (on ?disc ?from)
9          (clear ?disc) (clear ?to)
10         (not (= ?from ?to))
11     )
12     :effect (and
13       (clear ?from)
14       (on ?disc ?to)
15       (not (on ?disc ?from))
16       (not (clear ?to))
17     )
18   )
19 )
```

The PDDL *domain* file of the Tower of Hanoi is quite simple. Indeed, it consists of only one action (`move`) and only a few predicates. Firstly, the name given to this *domain* is `hanoi`. Then, there have been specified requirements as `:strips`, `:negative-preconditions` and `:equality`. After that, at line 3, there is the definition of all predicates involved in the PDDL *domain*. In particular, there are three predicates to describe if the top of a disk is `clear`, which disk is `on` top of another and, finally, which disk is `smaller` than another. Finally, there is the `move` action declaration with its parameters, its precondition formula and its effect formula.

3.1.2 The *problem* file

After having examined how a PDDL *domain* is defined, we can see the formulation of a PDDL *problem*. A PDDL *problem* is what a planner tries to solve. The *problem* file has the following format:

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
  )
```

At first glance, we can notice that the *problem* definition includes the specification of the domain to which it is related. Indeed, every problem is defined with respect to a precise *domain*. Then, there is the object list which could be typed or untyped. After that, there are the initial and goal specification, respectively. The former defines what is true at the beginning of the planning task and it consists of ground atoms, namely predicates instantiated with previously defined objects. Finally, the goal description represents the formula, consisting of instantiated predicates, that we would like to achieve and obtain as a final state. In the following, we show a simple example of PDDL *problem*.

Example 3.2. In this example, we show a possible PDDL *problem* for the Tower of Hanoi game for which we have shown the *domain* in the Example 3.1.

```
1  (define (problem hanoi-prob)
2    (:domain hanoi)
3    (:objects rod1 rod2 rod3 d1 d2 d3)
4    (:init
5       (smaller d1 rod1) (smaller d2 rod1) (smaller d3 rod1)
6       (smaller d1 rod2) (smaller d2 rod2) (smaller d3 rod2)
7       (smaller d1 rod3) (smaller d2 rod3) (smaller d3 rod3)
8       (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
9       (clear rod2) (clear rod3) (clear d1)
10      (on d3 rod1) (on d2 d3) (on d1 d2))
11   (:goal (and (on d3 rod3) (on d2 d3) (on d1 d2)))
12 )
```

At line 3, we have three rods and three disks. At the beginning, all instantiated predicates that are true are mentioned. If a predicate is not mentioned, it is considered to be false. In the initial situation there have been specified all possible movements with the `smaller` predicate, the disks are one on top of the other in ascending order on `rod1` whereas the other two rods are `clear`. In addition, the goal description is a conjuctive formula requiring disks on a stack on the `rod3`.

Once both PDDL *domain* and a *problem* are specified, they are given as input to planners.

3.2 Fully Observable Non Deterministic Planning

In this section, we formally define what *Fully Observable Non Deterministic Planning* (FOND) is giving some notions and definitions. Initially, we recall some concepts of "classical" planning while assuming the reader to be acquainted with basics of planning.

Given a PDDL specification with a *domain* and its corresponding *problem*, we would like to solve this specification in order to find a sequence of actions such that the goal formula holds true at the end of the execution. A *plan* is exactly that sequence of actions which leads the agent to achieve the goal starting from the initial state. Formally, we give the following definition.

Definition 3.1. A planning problem is defined as a tuple $\mathcal{P} = \langle \Sigma, s_0, g \rangle$, where:

- Σ is the state-transition system;
- s_0 is the initial state;
- g is the goal state.

Given the above Definition 3.1, we can formally define what a plan is.

Definition 3.2. A *plan* is any sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is a ground instance of an operator defined in the domain description.

Moreover, we have that:

Definition 3.3. A *plan* is a solution for $\mathcal{P} = \langle \Sigma, s_0, g \rangle$, if it is executable and achieves g.

Furthermore, a "classical" planning problem, just defined, is given under the assumptions of *fully observability* and *determinism*. In particular, the former means that the agent can always see the entire state of the environment whereas the latter means that the execution of an action is certain, namely any action that the agent takes uniquely determines its outcome.

Unlike the "classical" planning approach, in this thesis we focus on *Fully Observable Non Deterministic* (FOND) planning. Indeed, we continue relying on the *fully observability*, but losing the *determinism*. In other words,

in FOND planning we have the uncertainty on the outcome of an action execution. As anticipated in Section 3.1, the uncertainty of the outcome of an operator execution is syntactically expressed, in PDDL, with the keyword `oneof`. To better capture this concept, we give the following example.

Example 3.3. Here, we show as example the `put-on-block` operator of the FOND version of the well-known blocksworld PDDL *domain*.

```
(:action put-on-block
  :parameters (?b1 ?b2 - block)
  :precondition (and (holding ?b1) (clear ?b2))
  :effect (oneof (and (on ?b1 ?b2) (emptyhand) (clear ?b1)
                 (not (holding ?b1)) (not (clear ?b2)))
                 (and (on-table ?b1) (emptyhand) (clear ?b1)
                 (not (holding ?b1)))
          )
)
```

The effect of the `put-on-block` is non deterministic. Specifically, the action is executed every time the agent is holding a block and another block is clear on the top. The effect can be either that the block is put on top of the other block or that the block is put on table. This has to be intended as an aleatory event. Indeed, the agent does not control the operator execution result.

Additionally, a *non-deterministic* action a with effect $oneof(E_1, \ldots, E_n)$ can be intended as a set of *deterministic* actions $\{b_1, \ldots, b_n\}$, sharing the same precondition of a, but with effects $E_1, \ldots, E_n$, respectively. Hence, the application of action a turns out in the application of one of the actions b_i, chosen non-deterministically.

At this point, we can formally define the FOND planning. Following (Ghallab et al., 2004) and (Geffner and Bonet, 2013), we give the following definition:

Definition 3.4. A *non-deterministic domain* is a tuple $\mathcal{D} = \langle 2^{\mathcal{F}}, A, s_0, \varrho, \alpha \rangle$ where:

- $\mathcal{F}$ is a set of *fluents* (atomic propositions);
- A is a set of *actions* (atomic symbols);
- $2^{\mathcal{F}}$ is the set of states;
- s_0 is the initial state (initial assignment to fluents);

- $\alpha(s) \subseteq A$ represents *action preconditions*;
- $(s, a, s') \in \varrho$ with $a \in \alpha(s)$ represents *action effects* (including frame assumptions).

Such domain $\mathcal{D}$ is assumed to be represented compactly (e.g. in PDDL), therefore, considering the *size* of the domain as the cardinality of $\mathcal{F}$. Intuitively, the evolution of a non-deterministic domain is as follows: from a given state s, the agent chooses what action $a \in \alpha(s)$ to execute, then, the environment chooses a *successor state* s' with $(s, a, s') \in \varrho$. To this extent, planning can also be seen as a *game* between two players: the agent tries to force eventually reaching the goal no matter how the environment behaves. Moreover, the agent can execute an action having the knowledge of all history of states so far.

Now, we can define the meaning of solving a FOND planning problem on $\mathcal{D}$. A *trace* of $\mathcal{D}$ is a finite or infinite sequence $s_0, a_0, s_1, a_1, \ldots$ where s_0 is the initial state, $a_i \in \alpha(s_i)$ and $s_{i+1} \in \varrho(s_i, a_i)$ for each s_i, a_i in the trace.

Solutions to a FOND problem $\mathcal{P}$ are called *strategies* (or *policies*). A *strategy* π is defined as follows:

Definition 3.5. Given a FOND problem $\mathcal{P}$, a *strategy* π for $\mathcal{P}$ is a partial function defined as:

$$\pi : (2^{\mathcal{F}})^+ \to A \tag{1}$$

such that for every $u \in (2^{\mathcal{F}})^+$, if $\pi(u)$ is defined, then $\pi(u) \in \alpha(last(u))$, namely it selects applicable actions, whereas, if $\pi(u)$ is undefined, then $\pi(u) = \perp$.

A trace τ is *generated* by π (often called π-trace) if the following holds:

- if $s_0, a_0, \ldots, s_i, a_i$ is a prefix of τ, then $\pi(s_0, s_1, \ldots, s_i) = a_i$;
- if τ is finite, i.e. $\tau = s_0, a_0, \ldots, a_{n-1}, s_n$, then $\pi(s_0, s_1, \ldots, s_i) = \perp$.

For FOND planning problems, in (Cimatti et al., 2003), are defined different classes of solutions. Here we examine only two of them, namely *strong solution* and *strong cyclic solutions*. In the following, we give their formal definitions.

Definition 3.6. A *strong solution* is a strategy that is guaranteed to achieve the goal regardless of non-determinism.

Definition 3.7. *Strong cyclic solutions* guarantee goal reachability only under the assumption of *fairness*. In the presence of *fairness* it is supposed that all action outcomes, in a given state, would occur infinitely often.

Obviously, *strong cyclic solutions* are less restrictive than a *strong solution*. Indeed, as the name suggests, a strong cyclic solution may revisit states. However, in this thesis we will focus only on searching *strong solutions*. As final remark, when searching for a strong solution to a FOND problem we refer to FOND_{sp}.

In the next section, we will generalize the concept of solving FOND planning problems with extended temporal goals, describing the step by step encoding process of those temporal goals in the FOND domain, written in PDDL.

4 The FOND4LTL$_f$/PLTL$_f$ approach

As written in Section 2, planning with extended temporal goals has been considered over the representation of goals in classical planning to capture a richer class of plans where restrictions on the whole sequence of states must be satisfied as well. In particular, differently from classical planning, where the goal description can only be expressed as a propositional formula, in planning for extended temporal goals the goal description may have the same expressive power of the temporal logic in which the goal is specified. This enlarges the general view about planning. In other words, extended temporal goals specify desirable sequences of states and a plan exists if its execution yields one of these desirable sequences (Bacchus and Kabanza, 1998).

In this thesis, we propose a new approach, called FOND4LTL$_f$/PLTL$_f$, that uses LTL$_f$ and PLTL$_f$ formalisms as temporal logics for expressing extended goals. To better understand the powerful of planning with extended temporal goals we give the following example.

Example 4.1. Considering the well known `triangle tireworld` FOND planning task. The objective is to drive from one location to another, however while driving a tire may be going flat. If there is a spare tire in the location of the car, then the car can use it to fix the flat tire. The task in depicted in Figure 1, where there are locations arranged as a triangle, arrows representing roads and circles meaning that in a location there is a spare tire.

A possible classical goal can be $G = vehicleAt(l31)$, namely a propositional formula saying something only about what have to be true at the end of the execution. In the case of G, we exclusively require that the vehicle should be in location $l31$.

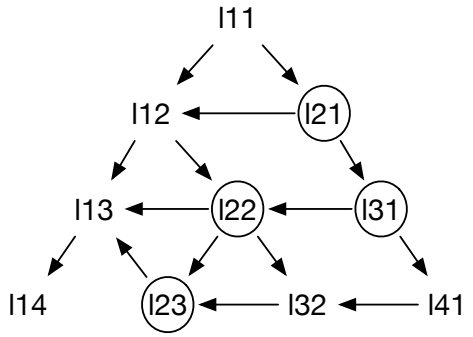

Figure 1: A possible Triangle Tireworld task. Locations marked with a circle have a spare tire, arrows represent possible directions

On the contrary, a goal specification expressed with temporal formalism such as PLTL$_f$ could be $\varphi = vehicleAt(l13) \wedge \Diamond(vehicleAt(l23))$. Such a specification requires to reach position $l13$, imposing the passage through position $l23$ before reaching the goal.

Planning for LTL$_f$ and PLTL$_f$ goals slightly changes the definitions given in Section 3.2. In the following, we give the modified definitions of the concepts seen before.

Definition 4.1. Given a domain $\mathcal{D}$ and an LTL$_f$/PLTL$_f$ formula φ over atoms $\mathcal{F} \cup \mathcal{A}$, a strategy π is a *strong solution* to $\mathcal{D}$ for goal φ, if every π-trace is finite and satisfies φ.

About complexity of FOND$_{sp}$, we have the following Theorems.

Theorem 4.1. *(De Giacomo and Rubin, 2018) Solving* FOND$_{sp}$ *for* LTL$_f$ *goals is:*

- EXPTIME-*complete in the size of the domain;*
- 2EXPTIME-*complete in the size of the goal.*

Furthermore, if the goal has the form of $\Diamond G$, i.e. is a reachability goal, the cost with respect to the goal becomes polynomial because it is just a propositional evaluation. If for a given LTL$_f$ goal the determinization step does not cause a state explosion, the complexity with respect to the goal is EXPTIME. On the contrary, if G is a PLTL$_f$ formula, from results in De Giacomo and Rubin (2018), we can only say that the complexity is 2EXPTIME in the size of the goal. Even though we remark that the investigation on

planning for PLTL$_f$ goals may have a computational advantage since PLTL$_f$ formulas can be reduced to DFA in single exponential time (vs. double-exponential time of LTL$_f$ formulas) (Chandra et al., 1981), computing the hardness is not obvious because we should evaluate the formula only after the $\Diamond$ operator.

4.1 Idea

Our FOND4LTL$_f$/PLTL$_f$ approach works as follows: given a non-deterministic planning domain $\mathcal{D}$, an initial state s_0 and an LTL$_f$ or PLTL$_f$ goal formula φ (whose symbols are ground predicates), we first obtain the corresponding DFA of the temporal formula through LTL$_f$2DFA, then, we encode such a DFA into the non-deterministic planning domain $\mathcal{D}$. As a result, we will have a new domain $\mathcal{D}'$ and a new problem P' that can be considered and solved as a classical FOND planning problem.

The new approach, carried out in this thesis, stems from the research in De Giacomo and Rubin (2018), that, basically, proposes automata-theoretic foundations of FOND planning for LTL$_f$ goals. In particular, they compute the cartesian product between the DFA corresponding to the domain $\mathcal{D}$ ($\mathcal{A}_\mathcal{D}$) and the DFA corresponding to φ ($\mathcal{A}_\varphi$), thus, solving a DFA game on $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$, i.e. find, if exists, a trace accepted by $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$.

However, unlike what has been done in De Giacomo and Rubin (2018), we split transitions containing the action and its effect in order to have them separately. The reason for this separation is that having both the action and its effect on the same transition is not suitable on a practical perspective. Hence, we have devised a solution in which we run $\mathcal{A}_\mathcal{D}$ and $\mathcal{A}_\varphi$ separately, but combining them into a single unique transition system. To achieve this, we move $\mathcal{A}_\mathcal{D}$ and $\mathcal{A}_\varphi$ alternatively by introducing an additional predicate, which we will call `turnDomain`, that is true when we should move $\mathcal{A}_\mathcal{D}$ and is false when we should move $\mathcal{A}_\varphi$. In the following, we give an example to better understand the solution put in place in this thesis.

Example 4.2. Let us consider the simplified version of the classical Yale shooting domain (Hanks and McDermott, 1986) as in De Giacomo and Rubin (2018), where we have that the turkey is either alive or not and the actions are shoot and wait with the obvious effects, but with a gun that can be faulty. Specifically, shooting with a (supposedly) working gun can either end in killing the turkey or in the turkey staying alive and the discovery that the gun is not working properly. On the other hand, shooting (with care) with a gun that does not work properly makes it work and kills the turkey. The cartesian product $\mathcal{A}_\mathcal{D} \times \mathcal{A}_\varphi$ with $\varphi = \Diamond\neg a$ is as follows:

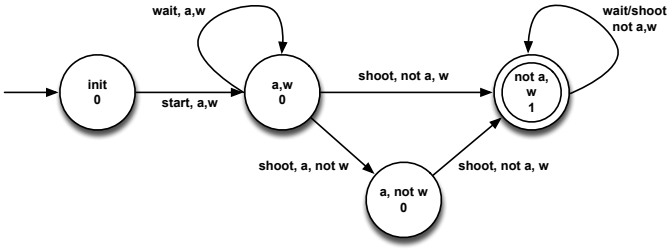

Figure 2: The DFA corresponding to $\mathcal{A}_{\mathcal{D}} \times \mathcal{A}_{\varphi}$. Symbol `a` stands for *alive* and `w` for *working*

As we can see in Figure 2, each transition reads both the action and its effect. This is not suitable for a practical implementation. Thus, we do not perform the cartesian product between the two automata. On the contrast, we build a transition system as in Figure 3.

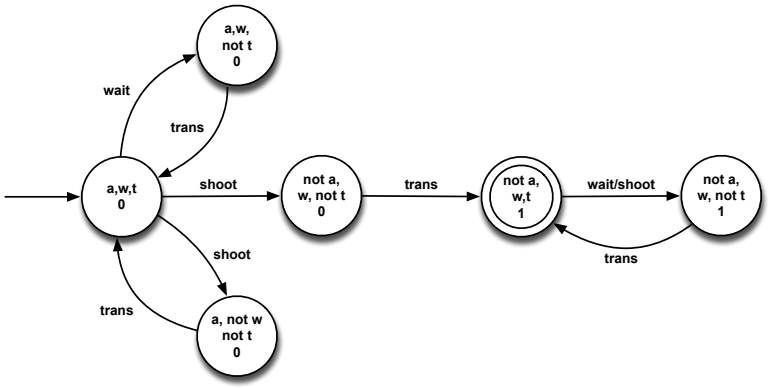

Figure 3: The new transition system corresponding to the Yale shooting domain. Symbol `a` stands for *alive*, `w` for *working* and `t` for *turnDomain*

The transition system, shown in Figure 3, expresses the new domain $\mathcal{D}'$ that has a perfect alternation of transitions. In particular, actions of the initial domain $\mathcal{D}$ alternates with a special action, that we called `trans`, representing the movement done by $\mathcal{A}_{\varphi}$. Moreover, it is important to notice the usage of the added predicate `turnDomain` allowing us to alternate between general actions and the new action `trans`.

In the next section, we will explain how the new domain $\mathcal{D}'$ and the new problem $\mathcal{P}'$ can be written in PDDL by showing the encoding of $\text{LTL}_f/\text{PLTL}_f$ goals in PDDL.

4.2 Encoding of LTL$_f$/PLTL$_f$ goals in PDDL

In this section, we describe the process of obtaining the new domain $\mathcal{D}'$ and the new problem $\mathcal{P}'$, both specified in PDDL. The original PDDL domain $\mathcal{D}$ and the associated original problem $\mathcal{P}$ change when introducing LTL$_f$/PLTL$_f$ goals. In particular, what changes is the way we encode our LTL$_f$/PLTL$_f$ formula in PDDL. Firstly, we employ our LTL$_f$2DFA tool to convert the given goal formula φ into the corresponding DFA. Then, we encode, in a specific way, the resulting DFA automaton in PDDL modifying the original domain $\mathcal{D}$ and problem $\mathcal{P}$.

Translation of DFAs in PDDL in Domain $\mathcal{D}$

In order to explain the translation technique of a DFA to PDDL, we assume to already have the DFA generated by LTL$_f$2DFA. We recall that such a DFA is formally defined as follows:

Definition 4.2. A DFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where:

- $\Sigma = \{a_0, a_1, \ldots, a_n\}$ is a finite set of symbols;
- $Q = \{q_0, q_1, \ldots, q_m\}$ is the finite set of states;
- $q_0 \in Q$ is the initial state;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $F \subseteq Q$ is the set of final states;

Specifically, since the automaton $\mathcal{A}$ corresponds to the goal formula φ, which has grounded predicates as symbols, we can represent them as $\{a_0(o_0, \ldots, o_j), \ldots, a_n(o_0, \ldots, o_w)\}$, where $o_0, \ldots, o_k \in \mathcal{O}$ and $0 \leqslant j, w \leqslant k$ represents objects present in the problem $\mathcal{P}$. To capture the general representation of φ in $\mathcal{D}$, we have to modify $\mathcal{A}$ to $\hat{\mathcal{A}}$ performing a transformation explained below. We give the following definitions.

Definition 4.3. $\hat{\mathcal{A}}$ is a tuple $\hat{\mathcal{A}} = \langle \hat{\Sigma}, \hat{Q}, \hat{q}_0, \hat{\delta}, \hat{F} \rangle$, where:

- $\hat{\Sigma} = \{\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_n\}$ is a finite set of symbols;
- $\hat{Q} = \{\hat{q}_0, \hat{q}_1, \ldots, \hat{q}_m\}$ is the finite set of states;
- $\hat{q}_0 \in \hat{Q}$ is the initial state;
- $\hat{\delta} : \hat{Q} \times \hat{\Sigma} \to \hat{Q}$ is the transition function;

- $\hat{F} \subseteq \hat{Q}$ is the set of final states;

Definition 4.4. Given the set of DFA symbols Σ, we define a mapping function m as follows:

$$m : \mathcal{O} \rightarrow \mathcal{V} \tag{2}$$

where $\mathcal{O}$ is the set of objects $\{o_0, \ldots, o_k\}$ and $\mathcal{V}$ is a set of variables $\{x_0, \ldots, x_k\}$

The transformation from $\mathcal{A}$ and $\hat{\mathcal{A}}$ is carried out with the mapping function m as follows:

- $\hat{\Sigma} = \{\hat{a}_0, \ldots, \hat{a}_n\}$, where $\hat{a}_i \doteq a_i(x_0, \ldots, x_j)$ and $x_0, \ldots, x_j \subseteq \mathcal{V}$;
- $\hat{Q} = \{\hat{q}_0, \ldots, \hat{q}_m\}$, where $\hat{q}_i \doteq q_i(x_0 \ldots, x_k)$.

Then, $\hat{q}_0, \hat{\delta}$ and $\hat{F}$ are modified accordingly.

Once the transformation is done, we have obtained a *parametric* DFA, which is a general representation with respect to the original one. After that, for representing the DFA transitions in the domain $\mathcal{D}$, we should encode the new *transition function* $\hat{\delta}$ into PDDL. To this extent, the $\hat{\delta}$ function is represented as a new PDDL operator, called `trans` having these properties:

- all variables in $\mathcal{V}$ are parameters;
- the negation of the `turnDomain` predicate is a precondition;
- effects represent the $\hat{\delta}$ function.

Moreover, effects are expressed as conditional effects. The general encoding would be as follows:

Action `trans`:
parameters: $(x_0, \ldots, x_k)$, where $x_i \in \mathcal{V}$
preconditions: $\neg turnDomain$
effects: when $(q_i(x_0, \ldots, x_k) \wedge \hat{a}_j)$ then $(\hat{\delta}(\hat{q}_i, \hat{a}_j) = q'_i(x_0, \ldots, x_k) \wedge (\neg q, \forall q \in \hat{Q} \text{ s.t. } q \neq q'_i) \wedge turnDomain)$, $\forall i, j : 0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n$

Additionally, in PDDL, especially in the effect formula of a conditional effect, we should specify that if the automaton is in a state, it is not in other states. This is captured by adding the negation of all other automaton states. In the following, we give an example showing the translation of DFAs to PDDL step-by-step.

Example 4.3. Let us consider the goal formula $\varphi = \Diamond(on(d3, rod3))$ for the Tower of Hanoi planning problem. The predicate `on` is instantiated on objects `d3` and `rod3`. By applying the mapping function m we have the corresponding variables and φ becomes $\varphi(x_1, x_2) = \Diamond(on(x_1, x_2))$, where we know that $x_1 = f(d_3)$ and $x_2 = f(rod_3)$. In this case, the modified DFA $\mathcal{A}'$ is depicted in Figure 4. At this point, consider the new DFA, the `trans`

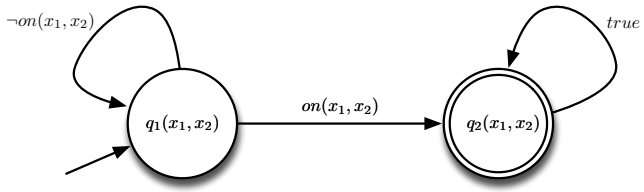

Figure 4: The parametric DFA corresponding to $\varphi(x_1, x_2) = \Diamond(on(x_1, x_2))$

operator built from that automaton is the following:

```
1 (:action trans
2   :parameters (?x1 ?x2)
3   :precondition (not (turnDomain))
4   :effect (and (when (and (q1 ?x1 ?x2) (not (on ?x1 ?x2)))
5                (and (q1 ?x1 ?x2) (not (q2 ?x1 ?x2)) (turnDomain))
6           (when (or (and (q1 ?x1 ?x2) (on ?x1 ?x2)) (q2 ?x1 ?x2))
7                (and (q2 ?x1 ?x2) (not (q1 ?x1 ?x2)) (turnDomain))
8   )
9 )
```

As just shown in Example 4.3, transitions, with source state and destination state, are encoded as conditional effects, where the condition formula includes source state and formula symbols whereas the effect formula includes the destination state, the negation of all other states and `turnDomain`. Moreover, in order to get a compact encoding of `trans` effects, conditional effects are brought together by destination state as happens, for instance, at line 6.

After the `trans` operator has been built, we change $\mathcal{D}$ as follows:

1. $\forall a \in A$: add `turnDomain` to $\alpha(s)$, i.e add `turnDomain` predicate to all actions precondition $\alpha(s)$;

2. $\forall a \in A$: add `(not (turnDomain))` to $(s, a, s') \in \varrho$ with $a \in \alpha(s)$, i.e add negated `turnDomain` to all actions effects (s, a, s');

3. add `trans` operator;

4. $\forall q' \in Q'$: add q' to predicates definition of $\mathcal{D}$, i.e. add all automaton state predicates to the domain predicates definition.

We have thus obtained the new domain $\mathcal{D}'$. In the following, we show an example.

Example 4.4. Let consider again the Triangle Tireworld scenario. The original PDDL domain is:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to)
      (not-flattire))
    :effect (oneof (and (vehicle-at ?to) (not (vehicle-at ?from)))
          (and (vehicle-at ?to) (not (vehicle-at ?from))
          (not (not-flattire))))
   )
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicle-at ?loc))
    :effect (and (not (spare-in ?loc)) (not-flattire))
   )
)
```

Now, consider a simple LTL_f formula $\varphi = \Diamond vehicleAt(l13)$. It requires that eventually the vehicle will be in location $l13$. The parametric DFA associated to $\varphi(x)$ is depicted in Figure 5.

Considering the DFA in Figure 5, the `trans` operator built from that automaton is the following:

```
(:action trans
  :parameters (?x - location)
  :precondition (not (turnDomain))
  :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
```

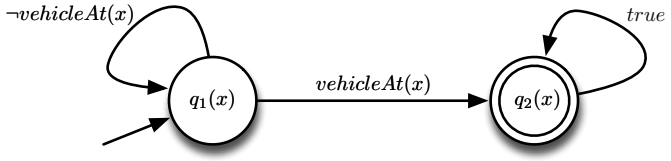

Figure 5: The parametric DFA corresponding to $\varphi(x) = \Diamond(vehicleAt(x))$

```
                (and (q1 ?x) (not (q2 ?x)) (turnDomain))
            (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
                (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

Finally, putting together all pieces and carrying out changes described above, we obtain the new domain $\mathcal{D}'$ as follows:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire)
               (q1 ?x - location)
               (q2 ?x - location)
               (turnDomain))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to)
      (not-flattire) (turnDomain))
    :effect (oneof (and (vehicle-at ?to) (not (vehicle-at ?from))
      (not (turnDomain)))
          (and (vehicle-at ?to) (not (vehicle-at ?from))
          (not (not-flattire)) (not (turnDomain))))
   )
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicle-at ?loc)
      (turnDomain))
    :effect (and (not (spare-in ?loc)) (not-flattire)
```

```
25          (not (turnDomain)))
26      )
27      (:action trans
28        :parameters (?x - location)
29        :precondition (not (turnDomain))
30        :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
31                    (and (q1 ?x) (not (q2 ?x)) (turnDomain))
32                (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
33                    (and (q2 ?x) (not (q1 ?x)) (turnDomain)))
34      )
35  )
```

Change in Problem $\mathcal{P}$

Concerning the planning problem $\mathcal{P}$, we completely discard the goal specification, whereas the initial state description is slightly modified. Moreover, the problem name, the associated domain name and all defined objects remain unchanged. We have to modify both the initial state and the goal state specifications to make them compliant with $\mathcal{D}'$, containing all changes introduced in the planning domain $\mathcal{D}$. To this extent, we formally define the new initial state as follows:

$$\text{Init: } s_0 \wedge \texttt{turnDomain} \wedge q_0 \tag{3}$$

where $q_0 \doteq \hat{q}_0(m^{-1}(x_0), \ldots, m^{-1}(x_k)) = q_0(o_0, \ldots, o_k)$. In other words, we put together the original initial specification s_0, the new predicate `turnDomain`, meaning that it is *true* at the beginning, and the initial state of the automaton instantiated on the objects of interest, i.e. those specified in the $\text{LTL}_f/\text{PLTL}_f$ formula φ.

On the other hand, the goal description is built from scratch as follows:

$$\text{Goal: } \texttt{turnDomain} \wedge (\bigvee_{q \in F} q) \tag{4}$$

where $q \doteq \hat{q}(m^{-1}(x_0), \ldots, m^{-1}(x_k)) = q(o_0, \ldots, o_k)$. In other words, we place together the `turnDomain` predicate, meaning that it must be *true* at the end of the execution, and the final state(s) of the automaton always instantiated on the objects of interest. Here, it is important to notice that if the automaton has two or more final states, they should be put in disjunction.

We can give the following example.

Example 4.5. Let consider again the Triangle Tireworld scenario, shown in the Example 4.4. The original PDDL domain is:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire))
  (:goal (vehicle-at l13))
)
```

Now, considering the same LTL_f formula $\varphi = \Diamond vehicleAt(l13)$, the object of interest is $l13$. Hence, we should evaluate our automaton states as $q_1(l13)$ and $q_2(l13)$.

Finally, putting together all pieces and carrying out changes described above, we obtain the new problem $\mathcal{P}'$ as follows:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire) (turnDomain) (q1 l13))
  (:goal (and (turnDomain) (q2 l13)))
)
```

As a remark, we will refer to the new goal specification as $\mathcal{G}'$.

Having examined the encoding of $\text{LTL}_f/\text{PLTL}_f$ goal formulas in PDDL, the resulting planning domain $\mathcal{D}'$ and problem $\mathcal{P}'$ represent a "classical" planning specification. In the next Section, we will see how we obtain a strong policy giving $\mathcal{D}'$ and $\mathcal{P}'$.

4.3 FOND Planners

In this Section, we talk about the state-of-art FOND planners and how they are employed within this thesis.

To begin with, thanks to our encoding process, we have reduced the problem of FOND planning for $\text{LTL}_f/\text{PLTL}_f$ goals to a "classical" FOND planning, which is essentially a *reachability* problem. We can state the following Theorem.

Theorem 4.2. *A strong policy π is a valid policy for $\mathcal{D}', \mathcal{G}'$ if and only if π is a valid policy for $\mathcal{D}, \varphi$.*

Given this Theorem, we can solve our original problem giving $\mathcal{D}'$ and $\mathcal{P}'$ as input to standard FOND planners. The main state-of-art FOND planners are:

- MBP and Gamer, which are OBDD[3]-based planners (Cimatti et al., 2003; Kissmann and Edelkamp, 2009)
- MyND and Grendel, which rely on explicit AND/OR graph search (Bercher, 2010; Ramirez and Sardina, 2014)
- PRP, NDP and FIP, which rely on classical algorithms (Kuter et al., 2008; Fu et al., 2011; Muise et al., 2012)
- FOND-SAT, which provides a SAT approach to FOND planning (Geffner and Geffner, 2018)

Although FOND planning is receiving an increasingly interest, the research on computational approaches has been recently reduced. Nevertheless, some planners performs well on different contexts of use. In our thesis, we are going to employ a customized version of FOND-SAT, the newest planner.

Secondly, although the description of many real-world planning problems involves the use of conditional effects, requiring full support of ADL by planners, those state-of-art planners, cited above, are still not able to fully handle such conditional effects. This represents a big limitation that can be surely deepened as a future work of this thesis. To this extent, we should first compile away conditional effects from the domain and, then, we can give it to a planner. Our proposal implementation is able to compile away simple conditional effects, namely those conditional effects that do not have nested formulas. Additionally, we have to compile away conditional effects of the `trans` operator upstream even though its representation with the employment of conditional effects is much more effective and compact. Luckily, this process consists of just splitting the operator in as many operators as the number of conditional effects present in the original action and adding

[3]OBDD stands for *Ordered Binary Decision Diagram*

the condition formula in the preconditions for each conditional effect. In the following, we make a clarifying example.

Example 4.6. The `trans` operator built in the Example 4.4 is:

```
(:action trans
  :parameters (?x - location)
  :precondition (not (turnDomain))
  :effect (and (when (and (q1 ?x) (not (vehicle-at ?x)))
               (and (q1 ?x) (not (q2 ?x)) (turnDomain))
          (when (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
               (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

As we can see, it contains only two conditional effects. Hence, we split this operator in two operators that we are going to call `trans-0` and `trans-1`, respectively. In particular, for each conditional effect the condition formula is added to the precondition and the effect formula is left in the effects. `trans-0` and `trans-1` are as follows:

```
(:action trans-0
  :parameters (?x - location)
  :precondition (and (and (q1 ?x) (not (vehicle-at ?x)))
    (not (turnDomain)))
  :effect (and (q1 ?x) (not (q2 ?x)) (turnDomain))
  )
)
(:action trans-1
  :parameters (?x - location)
  :precondition (and (or (and (q1 ?x) (vehicle-at ?x)) (q2 ?x))
    (not (turnDomain)))
  :effect (and (q2 ?x) (not (q1 ?x)) (turnDomain))
  )
)
```

At this point, having compiled away simple conditional effects from the modified domain $\mathcal{D}'$, we can finally describe how we have employed FOND-SAT in our thesis.

The main reasons why we have chosen FOND-SAT are the following:

- it is written in pure Python, hence we can easily integrate it in our Python implementation;

- it performs reasonably well;
- it outputs all policies building a transition system whose states are called *controller states*.

FOND-SAT takes also advantage of the parser and translation PDDL-to-SAS+ scripts from PRP. When FOND-SAT was developed, PRP's translation scripts could not handle disjunctive preconditions that may be present in our `trans` operators. As a result, we have modified FOND-SAT with the newest version of those scripts directly from PRP.

The usage of FOND-SAT is really simple. From its source folder, it is necessary to run the following command in the terminal:

```
python main.py -strong 1 -policy 1 /path-to/domain.pddl
/path-to/problem.pddl
```

The command simply executes the main module of FOND-SAT requiring to find strong policies and to print, if exists, the policy found. We feed FOND-SAT with our new domain $\mathcal{D}'$ and new problem $\mathcal{P}'$.

Once strong plans are found, FOND-SAT displays the policy in four sections as follows:

- Atom (CS): for each controller state it tells what predicates are true;
- (CS, Action with arguments): for each controller state it tells which actions can be applied
- (CS, Action name, CS): it tells for each controller state what action is applied in that state and the successor state;
- (CS1, CS2): it means that the controller can go from CS1 to CS2.

Now, we give an output example.

Example 4.7. The following result has been obtained running FOND-SAT with the Triangle Tireworld domain and problem with the LTL$_f$ goal $\varphi = \Diamond vehicleAt(l31)$. What follows is only the displayed policy.

```
1 ...
2 Trying with 7 states
3 Looking for strong plans: True
4 Fair actions: True
5 # Atoms: 18
6 # Actions: 26
```

```
SAT formula generation time = 0.052484
# Clauses = 11041
# Variables = 1225
Creating formula...
Done creating formula. Calling solver...
SAT solver called with 4096 MB and 3599 seconds
Done solver. Round time: 0.016456
Cumulated solver time: 0.055322
==========
==========
Controller -- CS = Controller State
==========
==========
Atom (CS)
 ----------
----------
Atom q1(l31) (n0)
Atom vehicleat(l11) (n0)
Atom not-flattire() (n0)
Atom spare-in(l21) (n0)
Atom turndomain() (n0)
----------
-NegatedAtom turndomain() (n1)
Atom q1(l31) (n1)
Atom vehicleat(l21) (n1)
Atom not-flattire() (n1)
----------
Atom q1(l31) (n2)
-NegatedAtom turndomain() (n2)
Atom spare-in(l21) (n2)
Atom vehicleat(l21) (n2)
----------
Atom turndomain() (n3)
Atom q1(l31) (n3)
Atom vehicleat(l21) (n3)
Atom not-flattire() (n3)
----------
Atom q1(l31) (n4)
Atom spare-in(l21) (n4)
Atom vehicleat(l21) (n4)
```

```
Atom turndomain() (n4)
----------
Atom q1(l31) (n5)
Atom vehicleat(l31) (n5)
-NegatedAtom turndomain() (n5)
----------
Atom turndomain() (ng)
Atom q2(l31) (ng)
==========
==========
(CS, Action with arguments)
 ----------
(n0,move-car_DETDUP_0(l11,l21))
(n0,move-car_DETDUP_1)
(n0,move-car_DETDUP_1(l11,l21))
(n0,move-car_DETDUP_0)
(n1,trans-0_v4)
(n1,trans-0_v4(l31))
(n2,trans-0_v4(l31))
(n2,trans-0_v4)
(n3,move-car_DETDUP_0(l21,l31))
(n3,move-car_DETDUP_0)
(n3,move-car_DETDUP_1(l21,l31))
(n3,move-car_DETDUP_1)
(n4,changetire(l21))
(n4,changetire)
(n5,trans-11)
(n5,trans-11(l31))
==========
==========
(CS, Action name, CS)
 ----------
(n0,move-car_DETDUP_0,n1)
(n0,move-car_DETDUP_1,n2)
(n1,trans-0_v4,n3)
(n2,trans-0_v4,n4)
(n3,move-car_DETDUP_0,n5)
(n3,move-car_DETDUP_1,n5)
(n4,changetire,n1)
(n5,trans-11,ng)
```

```
87  ==========
88  (CS, CS)
89  ----------
90  (n2,n4)
91  (n5,ng)
92  (n3,n5)
93  (n1,n3)
94  (n0,n2)
95  (n4,n1)
96  (n0,n1)
97  ==========
98  Solved with 7 states
99  Elapsed total time (s): 0.288035
100 Elapsed solver time (s): 0.055322
101 Elapsed solver time (s): [0.0052, 0.0059, 0.007, 0.009, 0.011, 0.016]
102 Looking for strong plans: True
103 Fair actions: True
104 Done
```

As we can see, operators names are changed due to internal arrangements made by FOND-SAT needed to handle both non determinism and disjunctive preconditions. Here, it is important to observe that, as we expected, there is an alternation of action executions between the original domain actions and the `trans` operators. Finally, the transition system built by FOND-SAT has $n0$ as initial state and ng as final state. If strong plans are found, it means that every path from $n0$ to ng is a valid plan. We will better explain this later in Section 5.

In the following section, we will report results obtained through our practical implementation, called FOND4LTL$_f$/PLTL$_f$, that automates all the process illustrated in this Section.

5 Results

In this section, we show an execution of the FOND4LTL$_f$/PLTL$_f$ tool as example of result. In particular, we show an execution involving a PLTL$_f$ goal. We go step-by-step through the solution and we pay attention to the `trans` operator.

We remind the reader that FOND$_{sp}$ planning for PLTL$_f$ goals is interpreted as reaching a final state such that the history leading to such a state

satisfies the given PLTL$_f$ formula. For instance, here we show an execution of the FOND4LTL$_f$/PLTL$_f$ tool on the Triangle Tireworld planning task partly illustrated in Example 4.4. Indeed, the original domain is the same of the one in Example 4.4 whereas the original initial state is as follows:

```
(define (problem triangle-tire-1)
  (:domain triangle-tire)
  (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
  (:init (vehicle-at l11)
    (road l11 l12) (road l12 l13) (road l11 l21) (road l12 l22)
    (road l21 l12) (road l22 l13) (road l21 l31) (road l31 l22)
    (spare-in l21) (spare-in l22) (spare-in l31)
    (not-flattire))
  (:goal (vehicle-at l13))
)
```

In this case, we choose the PLTL$_f$ goal formula $\varphi = vehicleAt(l13) \wedge \Diamond(vehicleAt(l23))$. Such a formula means *reach location l13 passing through location l23 at least once*. The DFA corresponding to φ, generated with LTL$_f$2DFA, is depicted in Figure 6.

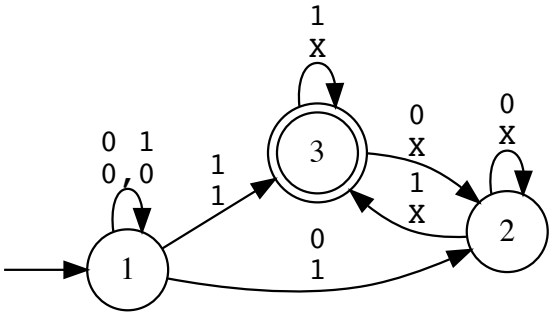

Figure 6: The DFA corresponding to φ

After the execution of FOND4LTL$_f$/PLTL$_f$, we obtain the following:

1. a new planning domain;
2. a new planning problem;
3. a transition system showing all policies, if found.

Firstly, the new domain $\mathcal{D}'$ is:

```
(define (domain triangle-tire)
  (:requirements :typing :strips :non-deterministic)
  (:types location)
  (:predicates (vehicleat ?loc - location) (spare-in ?loc - location)
  (road ?from - location ?to - location) (not-flattire) (turnDomain)
  (q2 ?loc30 - location ?loc53 - location)
  (q1 ?loc30 - location ?loc53 - location)
  (q3 ?loc30 - location ?loc53 - location))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicleat ?from) (road ?from ?to)
    (not-flattire) (turnDomain))
    :effect (and (oneof (and (vehicleat ?to)
    (not (vehicleat ?from))) (and (vehicleat ?to)
    (not (vehicleat ?from)) (not (not-flattire))))
    (not (turnDomain)))
  )
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicleat ?loc)
    (turnDomain))
    :effect (and (not (spare-in ?loc)) (not-flattire)
    (not (turnDomain)))
  )
  (:action trans-0
    :parameters (?loc30 - location ?loc53 - location)
    :precondition (and (or (and (q1 ?loc30 ?loc53)
    (not (vehicleat ?loc30)) (vehicleat ?loc53))
    (and (q2 ?loc30 ?loc53) (not (vehicleat ?loc30)))
    (and (q3 ?loc30 ?loc53) (not (vehicleat ?loc30))))
    (not (turnDomain)))
    :effect (and (q2 ?loc30 ?loc53)
    (not (q1 ?loc30 ?loc53)) (not (q3 ?loc30 ?loc53))
    (turnDomain))
  )
  (:action trans-1
    :parameters (?loc30 - location ?loc53 - location)
    :precondition (and (or (and (q1 ?loc30 ?loc53)
    (not (vehicleat ?loc30)) (not (vehicleat ?loc53)))
    (and (q1 ?loc30 ?loc53) (vehicleat ?loc30)
```

```
41      (not (vehicleat ?loc53)))) (not (turnDomain)))
42      :effect (and (q1 ?loc30 ?loc53)
43      (not (q2 ?loc30 ?loc53)) (not (q3 ?loc30 ?loc53))
44      (turnDomain))
45    )
46    (:action trans-2
47      :parameters (?loc30 - location ?loc53 - location)
48      :precondition (and (or (and (q1 ?loc30 ?loc53)
49      (vehicleat ?loc30) (vehicleat ?loc53))
50      (and (q2 ?loc30 ?loc53) (vehicleat ?loc30))
51      (and (q3 ?loc30 ?loc53) (vehicleat ?loc30)))
52      (not (turnDomain)))
53      :effect (and (q3 ?loc30 ?loc53)
54      (not (q2 ?loc30 ?loc53)) (not (q1 ?loc30 ?loc53))
55      (turnDomain))
56    )
57  )
```

Secondly, the new problem $\mathcal{P}'$ is:

```
1  (define (problem triangle-tire-1)
2          (:domain triangle-tire)
3          (:objects l11 l12 l13 l21 l22 l23 l31 l32 l33 - location)
4          (:init (not-flattire) (q1 l13 l23) (road l11 l12)(road l11 l21)
5          (road l12 l13) (road l12 l22) (road l21 l12) (road l21 l31)
6          (road l22 l13) (road l22 l23) (road l23 l13) (road l31 l22)
7          (spare-in l21) (spare-in l22) (spare-in l23) (spare-in l31)
8          (turnDomain) (vehicleat l11))
9  (:goal (and (q3 l13 l23) (turnDomain)))
10 )
```

Finally, feeding FOND-SAT with $\mathcal{D}'$ and $\mathcal{P}'$ we obtain the following transition system depicted in Figure 7, thanks to another Python script, developed in this thesis, for converting a written policy into a graph.

A plan is whatever path from $n0$ leading to state ng. Moreover, as we can see from Figure 7, there is a perfect alternation between domain's actions and the `trans` action. Additionally, the above-mentioned script could also remove transitions involving the `trans` action getting the final plan. We can see it in Figure 8.

6 Summary

In this report, we have faced the problem of FOND Planning for $\text{LTL}_f/\text{PLTL}_f$ goals. In particular, we have proposed a new solution, called $\text{FOND4LTL}_f/\text{PLTL}_f$, that essentially reduces the problem to a "classical" FOND planning problem. This has been possible thanks to our LTL_f2DFA Python tool which has been employed for the encoding of temporally extended goals into standard PDDL. Finally, we have seen examples of execution results of the $\text{FOND4LTL}_f/\text{PLTL}_f$ tool.

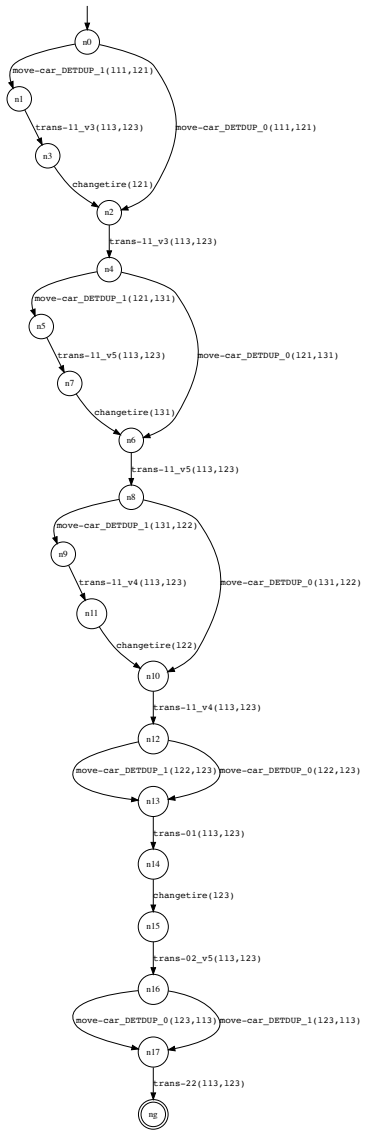

Figure 7: The transition system showing policies found with the `trans` operator

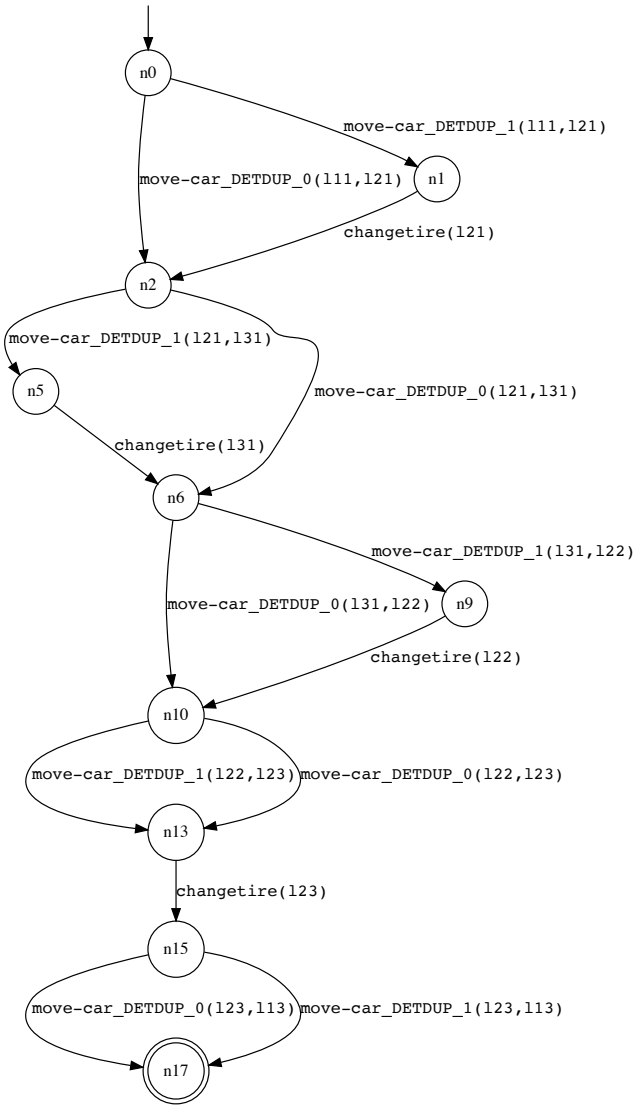

Figure 8: The transition system showing policies found without the `trans` operator

References

Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

Pascal Bercher. Pattern database heuristics for fully observable nondeterministic planning. 2010.

Alberto Camacho, Eleni Triantafillou, Christian J Muise, Jorge A Baier, and Sheila A McIlraith. Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces. In *AAAI*, pages 3716–3724, 2017.

AK Chandra, DC Kozen, LJ Stockmeyer, and J Alternation. Acm, vol. 28 (1), 1981.

Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

Giuseppe De Giacomo and Sasha Rubin. Automata-theoretic foundations of fond planning for ltlf and ldlf goals. In *IJCAI*, pages 4729–4735, 2018.

Patrick Doherty and Jonas Kvarnstram. Talplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95, 2001.

Jicheng Fu, Vincent Ng, Farokh B Bastani, I-Ling Yen, et al. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1949, 2011.

Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1):1–141, 2013.

Tomas Geffner and Hector Geffner. Compact policies for fully-observable non-deterministic planning as sat. *arXiv preprint arXiv:1806.09455*, 2018.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

Steve Hanks and Drew V McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *AAAI*, volume 86, pages 328–333, 1986.

Peter Kissmann and Stefan Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *Annual Conference on Artificial Intelligence*, pages 1–8. Springer, 2009.

Ugur Kuter, Dana Nau, Elnatan Reisner, and Robert P Goldman. Using classical planners to solve nondeterministic planning problems. In *Proceedings of the Eighteenth International Conference on International Conference on Automated Planning and Scheduling*, pages 190–197. AAAI Press, 2008.

Christian J Muise, Sheila A McIlraith, and J Christopher Beck. Improved non-deterministic planning by exploiting state relevance. In *ICAPS*, 2012.

Miquel Ramirez and Sebastian Sardina. Directed fixed-point regression-based planning for non-deterministic domains. In *ICAPS*, 2014.