PDDL to DFA: A Symbolic Transformation for Effective Reasoning

Giuseppe De Giacomo
University of Oxford, UK

Antonio Di Stasio
City St George's, University of London, UK

Gianmarco Parretti
La Sapienza University of Rome, Italy

Abstract

LTL$_f$ reactive synthesis under environment specifications, which concerns the automated generation of strategies enforcing logical specifications, has emerged as a powerful technique for developing autonomous AI systems. It shares many similarities with Fully Observable Nondeterministic (FOND) planning. In particular, nondeterministic domains can be expressed as LTL$_f$ environment specifications. However, this is not needed since nondeterministic domains can be transformed into deterministic finite-state automata (DFA) to be used directly in the synthesis process. In this paper, we present a practical symbolic technique for translating domains expressed in Planning Domain Definition Language (PDDL) into DFAs. The technique allows for the integration of the planning domain, reduced to DFA in a symbolic form, into current symbolic LTL$_f$ synthesis tools. We implemented our technique in a new tool, PDDL2DFA, and applied it to solve FOND planning by using state-of-the-art reactive synthesis techniques in a tool called SYFT4FOND. Our empirical results confirm the effectiveness of our approach.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Logic and verification; Computing methodologies → Symbolic and algebraic manipulation

Keywords and phrases Fully Observable Nondeterministic Planning, Linear Temporal Logics on finite traces, Reactive Synthesis, DFA

Digital Object Identifier 10.4230/LIPIcs.TIME.2025.7

Supplementary Material *Software (Source Code)*: `https://github.com/GianmarcoDIAG/syft4fond` [33]

1 Introduction

In recent years there has been a growing interest in applying Formal Methods techniques to Artificial Intelligence in order to develop autonomous AI systems that can operate effectively in dynamic and complex environments.

These techniques include reactive synthesis, which concerns the automated generation of *winning strategies* that enforce requirements given in the form of logical specifications [16, 34]. Specifically, we consider reactive synthesis for specifications in Linear Temporal Logic on Finite Traces (LTL$_f$) [26, 27], which maintains the syntax of LTL [35], the formalism typically used to express complex dynamic properties in Formal Methods [8], but it is interpreted on finite traces.

A key aspect shared by all synthesis work in AI is the need for a model of the environment in which the agent acts. In fully observable nondeterministic (FOND) planning [18, 17], such a model is given as a state-based domain that specifies, in each state, how the environment enacts the (possibly nondeterministic) effects of agent actions. In its most common form, FOND planning involves computing a *strong plan* that guarantees reaching one of the goal states independently of the nondeterminism in the domain, thus sharing many similarities with LTL$_f$ reactive synthesis [27, 15]. Specifically, a FOND domain represented with functions and sets can be expressed as an LTL$_f$ environment specification and transformed into a deterministic finite-state automaton (DFA) that accepts the traces consistent with the domain specification [25, 1, 6, 23]. It follows that FOND planning can be reduced to synthesizing a winning strategy over a DFA game [25, 23], as LTL$_f$ reactive synthesis [27].

In practice, FOND domains are often specified in a compact language like the Planning Domain Definition Language (PDDL) [31], which has been extensively used in planning competitions[1]. However, while how to transform a FOND domain into DFA is well-known in theory, how to effectively transform PDDL into DFA in practice is still open to further investigation.

In this paper, we present an effective technique for transforming PDDL into a symbolic DFA with transitions and final states represented as Boolean functions encoded by using Binary Decision Diagrams (BDDs) [13]. This technique allows for the integration of FOND domains into symbolic LTL$_f$ synthesis tools, which are known for their scalability and efficiency [39, 10, 9, 37]. The construction process involves representing the nondeterminism in the domain through suitable *agent actions* and *environment reactions*. Once this representation is established, the symbolic DFA of the domain can be efficiently constructed by manipulating BDDs. Our technique has the notable property that, while worst-case exponential in the size of the input domain, it is often polynomial due to its concise representation of the nondeterminism in the domain through a compact set of environment reactions.

We implemented our method in a new tool, PDDL2DFA, and applied it to devise a reduction of FOND planning into reactive synthesis (optimal wrt complexity of FOND planning, i.e., EXPTIME-complete [17]) in a tool called SYFT4FOND. We applied this construction to various case studies, including the classic blocks world, blocks world extended, an elevator system, and two navigation environments. Our empirical results show the performance of our approach in these different cases. Specifically, our technique successfully constructs the DFA for a considerable number of instances and solves the synthesis problem for a reasonable number of them, showing the practical feasibility of reducing planning to synthesis.

Our approach takes a step towards integrating planning and synthesis more closely and serves as a promising starting point for future research.

2 Preliminaries

Notations. A *trace* over an alphabet of symbols Σ is a finite or infinite sequence of elements from Σ. The empty trace is denoted λ. Traces are indexed starting at zero, and we write $\pi = \pi_0\pi_1\cdots$. For a finite trace π, let $\mathsf{lst}(\pi)$ denote the index of the last element of π, i.e., $\mathsf{lst}(\pi) = |\pi| - 1$.

2.1 Linear Temporal Logic on finite traces (LTL$_f$)

LTL$_f$ is a variant of LTL interpreted over *finite traces* [26] instead of *infinite traces* [35]. LTL$_f$ has the same syntax as LTL. Given a set AP of atomic propositions (aka atoms), the LTL$_f$ formulas over AP are generated by the following grammar:

$$\varphi ::= a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \Circle\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

Where $a \in AP$. Here $\Circle$ (*Next*) and $\mathcal{U}$ (*Until*) are temporal operators. We use standard Boolean abbreviations such as $\vee$ (or), $\supset$ (implies), *true* and *false*. Moreover, we define the following abbreviations: $\CIRCLE\varphi \equiv \neg\Circle\neg\varphi$ (*Weak Next*), $\Diamond\varphi \equiv true\,\mathcal{U}\,\varphi$ (*Eventually*), and

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

$\Box\varphi \equiv \neg\Diamond\neg\varphi$ (*Always*). The size of φ, written $|\varphi|$, is the number of its subformulas. Formulas are interpreted over finite traces π over the alphabet $\Sigma = 2^{AP}$, i.e., the alphabet consisting of the propositional interpretations of the atoms. Thus, for $0 \leq i \leq \mathsf{lst}(\pi)$, $\pi_i \in 2^{AP}$ is the i-th interpretation of π. That an LTL$_f$ formula φ *holds* at instant $i \leq \mathsf{lst}(\pi)$, written $\pi, i \models \varphi$, is defined inductively:

- $\pi, i \models a$ iff $a \in \pi_i$ (for $a \in AP$);
- $\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$;
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$;
- $\pi, i \models \mathsf{O}\varphi$ iff $i < \mathsf{lst}(\pi)$ and $\pi, i+1 \models \varphi$;
- $\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ iff $\exists j$ such that $i \leq j \leq \mathsf{lst}(\pi)$ and $\pi, j \models \varphi_2$, and $\forall k, i \leq k < j$ we have that $\pi, k \models \varphi_1$.

We say that π *satisfies* φ, written $\pi \models \varphi$, if $\pi, 0 \models \varphi$.

2.2 Deterministic Finite Automata

A deterministic finite automaton (DFA) is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$, where: Σ is a finite input alphabet; Q is a finite set of states; $q_0 \in Q$ is the initial state; $\delta : Q \times \Sigma \to Q$ is the transition function; and $F \subseteq Q$ is the set of final states. The size of $\mathcal{A}$ is $|Q|$. Given a finite trace $\alpha = \alpha_0\alpha_1 \ldots \alpha_n$ over Σ, we extend δ to be a function $\delta : Q \times \Sigma^* \to Q$ as follows: $\delta(q, \lambda) = q$, and, if $q_n = \delta(q, \alpha_0 \ldots \alpha_{n-1})$, then $\delta(q, \alpha_0 \ldots \alpha_n) = \delta(q_n, \alpha_n)$. A trace α is *accepted* by $\mathcal{A}$ if $\delta(q_0, \alpha) \in F$. The *language* of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of traces that the automaton accepts.

▶ **Theorem 1** ([26]). *Given an* LTL$_f$ *formula* φ*, we can build a* DFA *of at most doubly-exponential size in* $|\varphi|$ *whose language is the set of traces satisfying* φ*.*

2.3 LTL$_f$ Reactive Synthesis

LTL$_f$ *reactive synthesis* [27] concerns finding a strategy to satisfy an LTL$_f$ *goal* specification. Goals are expressed as LTL$_f$ formulas over $AP = \mathcal{X} \cup \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are disjoint sets of variables. Intuitively, $\mathcal{X}$ (resp. $\mathcal{Y}$) is under the environment's (resp. agent's) control. Traces over $\Sigma = 2^{\mathcal{X} \cup \mathcal{Y}}$ will be denoted $\pi = (X_0 \cup Y_0)(X_1 \cup Y_1) \ldots$ where $X_i \subseteq \mathcal{X}$ and $Y_i \subseteq \mathcal{Y}$ for every i. Infinite traces of this form are also called *plays*.

An *agent strategy* is a function $\sigma : (2^{\mathcal{X}})^* \to 2^{\mathcal{Y}}$ mapping sequences of environment moves to an agent move. The domain of σ includes the empty sequence λ as we assumed that the agent moves first. A trace π is σ-consistent if $Y_0 = \sigma(\lambda)$ and $Y_{j+1} = \sigma(X_0 \cdots X_j)$ for every $j \geq 0$. Let φ be an LTL$_f$ formula over $\mathcal{X} \cup \mathcal{Y}$. An agent strategy σ is *winning* for (aka *enforces*) φ if, for every play π that is σ-consistent, some finite prefix of π satisfies φ. LTL$_f$ reactive synthesis is the problem of finding an agent strategy σ that enforces φ, if one exists, and is 2EXPTIME-complete in the size of φ [27].

In many AI applications the agent has some knowledge about how the environment works, which it can exploit to enforce the goal [2]. This knowledge can be expressed by an LTL$_f$ formula $\mathcal{E}$ over $\mathcal{X} \cup \mathcal{Y}$, which we call *environment specification*. In the synthesis of winning strategies, we can intuitively see $\mathcal{E}$ as restricting traces of interest to those satisfying $\mathcal{E}$. In this setting, synthesis amounts to computing a strategy that enforces the implication $\mathcal{E} \supset \varphi$, if one exists.

2.4 DFA Games

A DFA game is a DFA $\mathcal{G} = (\Sigma, Q, q_0, \delta, F)$ with input alphabet $\Sigma = 2^{\mathcal{X} \cup \mathcal{Y}}$. The notions of agent strategy and play defined above also apply to DFA games. A play is *winning* if it contains a finite prefix that is accepted by the DFA. An agent strategy is *winning* if, for every play π that is σ-consistent, π is winning. That is, the agent wins the game if it can force the play to visit the set of final states at least once. The *winning region* is the set of states $q \in Q$ for which the agent has a winning strategy in the game $\mathcal{G}'$, where $\mathcal{G}' = (2^{\mathcal{X} \cup \mathcal{Y}}, Q, q, \delta, F)$, i.e., the same game as $\mathcal{G}$, but with initial state q. *Solving* a DFA game is the problem of computing the agent winning region and a winning strategy, if one exists. DFA games can be solved in polynomial time by a backward-induction algorithm that performs a fixpoint computation over the state space of the game [7]. Synthesis of an LTL$_f$ formula φ can be reduced in doubly-exponential time to solve the DFA game $\mathcal{G}_\varphi$ corresponding to φ [27].

Solving LTL$_f$ synthesis reduces to solving a reachability game over the DFA corresponding to the LTL$_f$ specification. The procedure for solving LTL$_f$ synthesis is detailed in Algorithm 1.

Algorithm 1 LTL$_f$ Synthesis.

```
Input: LTL$_f$ formula $\varphi$
Output: strategy $\sigma_{ag}$ realizing $\varphi$;
1: Compute the corresponding NFA $\mathcal{A}_\varphi$;
2: Determinize $\mathcal{A}_\varphi$ into a DFA $\mathcal{B}_\varphi$;
3: Solve the reachability game over $\mathcal{B}_\varphi$.
```

2.5 Symbolic Synthesis

We consider the DFA representation described above as an explicit-state representation. Instead, we are able to represent a DFA more compactly in a symbolic way by using a logarithmic number of propositions to encode the state space [40]. Formally, the *symbolic representation* of a DFA $\mathcal{A} = (2^{\mathcal{X} \cup \mathcal{Y}}, Q, q_0, \delta, F)$ is a tuple $\mathcal{A}^s = (\mathcal{X}, \mathcal{Y}, \mathcal{Z}, Z_0, \eta, f)$, where: $\mathcal{Z}$ is a set of *state variables* such that $|\mathcal{Z}| = \lceil \log |Q| \rceil$, and every state $q \in Q$ corresponds to an interpretation $Z \in 2^{\mathcal{Z}}$; $Z_0 \in 2^{\mathcal{Z}}$ is the interpretation corresponding to the initial state q_0; $\eta\colon 2^{\mathcal{Z}} \times 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \to 2^{\mathcal{Z}}$ is a Boolean function such that $\eta(Z, X, Y) = Z'$ if and only if Z is the interpretation of a state q and Z' is the interpretation of the state $\delta(q, X \cup Y)$; and f is a Boolean function over $\mathcal{Z}$ such that $f(Z) = 1$ if and only if Z is the interpretation corresponding to a state $q \in F$. Note that the transition function η can be represented by an indexed family consisting of a Boolean formula η_z for each state variable $z \in \mathcal{Z}$, which when evaluated over an assignment to $\mathcal{Z} \cup \mathcal{X} \cup \mathcal{Y}$ returns the next assignment to z.

A symbolic DFA game can be solved by performing a least fixpoint computation over two Boolean formulas w over $\mathcal{Z}$ and t over $\mathcal{Z} \cup \mathcal{Y}$ which represent the winning region and winning states with agent moves such that, regardless of how the environment behaves, the agent reaches the final states, respectively [40]. Specifically, w and t are initialized as $w_0(\mathcal{Z}) = f(\mathcal{Z})$ and $t_0(\mathcal{Z}, \mathcal{Y}) = f(\mathcal{Z})$, since every final state is a winning state. Note that t_0 is independent of the propositions from $\mathcal{Y}$, since once the play reaches the final states, the agent can do whatever it wants. Then, t_{i+1} and w_{i+1} are constructed as follows:

$$
\begin{aligned}
&t_{i+1}(Z, Y) = t_i(Z, Y) \lor (\neg w_i(Z) \land \forall X. w_i(\eta(X, Y, Z))) \\
&w_{i+1}(Z) = \exists Y. t_{i+1}(Z, Y)
\end{aligned}
$$

The computation reaches a fixpoint when $w_{i+1} \equiv w_i$. When the fixpoint is reached, no more states will be added, and all winning states have been collected. By evaluating Z_0 on w_{i+1} we can determine if there exists a winning strategy. If that is the case, t_{i+1} can be used to compute a uniform positional winning strategy through the mechanism of Boolean synthesis [28].

2.6 Fully Observable Non-Deterministic (FOND) Planning

Following [23], we define a FOND domain as a tuple $\mathcal{D} = (2^{\mathcal{F}}, s_0, \mathit{Act}, \mathit{React}, \alpha, \beta, \delta)$, where: $\mathcal{F}$ is a finite set of fluents, $|\mathcal{F}|$ is the size of $\mathcal{D}$, and $2^{\mathcal{F}}$ is the state space; Act and React are finite sets of agent actions and environment reactions, respectively; $\alpha : 2^{\mathcal{F}} \to 2^{\mathit{Act}}$ is a function denoting agent action preconditions; $\beta : 2^{\mathcal{F}} \times \mathit{Act} \to 2^{\mathit{React}}$ is a function denoting environment reaction preconditions; and $\delta : 2^{\mathcal{F}} \times \mathit{Act} \times \mathit{React} \to 2^{\mathcal{F}}$ is the transition function such that $\delta(s, a, r)$ is defined if and only if $a \in \alpha(s)$ and $r \in \beta(s, a)$. We assume that planning domains satisfy the properties of:

- *Existence of agent action*: $\forall s \in 2^{\mathcal{F}}.\exists a \in \alpha(s)$;
- *Existence of environment reaction*: $\forall s \in 2^{\mathcal{F}}.\forall a \in \alpha(s).\exists r \in \beta(s, a)$;
- *Uniqueness of environment reaction*:
  $\forall s \in 2^{\mathcal{F}}.\forall a \in \alpha(s).\forall r_1, r_2 \in \beta(s, a).\delta(s, a, r_1) = \delta(s, a, r_2) \supset r_1 = r_2$.

With these properties, inspired by [22], we can capture classical FOND domains [18, 29] by explicitly introducing environment reactions corresponding to nondeterministic effects of agent actions.

A *state trace* of $\mathcal{D}$ is a finite sequence $\tau = s_0 \cdots s_n$ of states such that s_0 is the initial state of $\mathcal{D}$ and, for every $i < n$, there exists an agent action $a_i \in \alpha(s_i)$ and an environment reaction $r_i \in \beta(s_i, a_i)$ such that $s_{i+1} = \delta(s_i, a_i, r_i)$. A *plan* is a partial function $\kappa : 2^{\mathcal{F}} \to \mathit{Act}$ such that, if $\kappa(s)$ is defined, then $\kappa(s) \in \alpha(s)$. A plan terminates its execution in states where, being a partial function, it specifies no action. A state trace $\tau = s_0 \cdots s_n$ is κ-consistent if: (i) s_0 is the initial state of $\mathcal{D}$; (ii) for every $i < n$, $s_{i+1} = \delta(s_i, a_i, r_i)$ for $a_i = \kappa(s_i)$ and some $r_i \in \beta(s_i, a_i)$; and (iii) $\kappa(s_n)$ is undefined.

FOND (strong) planning concerns finding a plan to satisfy a goal regardless of the nondeterminism in the domain, called *strong plan*. A goal G is a conjunction of fluents and negations of fluents. Given a goal G and a domain $\mathcal{D}$, a plan κ is strong for G in $\mathcal{D}$ if, for every state trace $\tau = s_0, \cdots, s_n$ that is κ-consistent, $s_n \models G$. Formally, FOND planning is the problem of finding a strong plan for G in $\mathcal{D}$, if one exists. FOND planning is EXPTIME-complete in the size of $\mathcal{D}$ [17].

In this paper, we always assume that we have FOND domains expressed in PDDL [31], i.e., the fluents defining the states of the domain are predicates over objects. We write `predicate/k` to specify that k objects participate in the `predicate` relation. Predicates, actions, and action preconditions are specified in first-order syntax in a `domain.pddl` file, whereas the initial state, goal, and objects, are usually specified in a separate `problem.pddl` file.

3 PDDL to Symbolic DFA

In this paper, we present a technique for transforming PDDL into symbolic DFA. A naive approach is to translate PDDL into LTL$_f$ [2] and build its corresponding symbolic DFA. However, this approach is limited by the doubly-exponential blow-up resulting from transforming LTL$_f$ formulas in DFAs [26]. Instead, our technique ensures only a single-exponential blowup in the number of fluents.

Our transformation is based on representing the nondeterminism in the domain with agent actions and environment reactions, as described in Section 2. Once such a representation is established, the symbolic DFA of the domain can be easily built by constructing suitable Boolean formulas.

The core idea to represent the nondeterminism in the domain with agent actions and environment reactions is as follows. For every state of the domain $s \in 2^{\mathcal{F}}$ and possible nondeterministic effect $s_1, \cdots, s_n$ of an agent action $a \in \alpha(s)$ (specified in its `oneof` clause), we introduce an environment reaction $r_1, \cdots, r_n$ such that, for every $i < n$, we have $\delta(s, a, r_i) = s_i$ and $r_i \in \beta(s, a)$. That is, r_i represents the i-th nondeterministic effect of applying action a in s. Applying this construction to every state $s \in 2^{\mathcal{F}}$ and agent action $a \in Act$ generates a planning domain $\mathcal{D} = (2^{\mathcal{F}}, s_0, Act, React, \alpha, \beta, \delta)$ as detailed in Algorithm 2

Algorithm 2 PDDL2ACTSANDREACTS(`domain.pddl`, `problem.pddl`).

```
Require: PDDL description of planning domain as domain.pddl and problem.pddl
Ensure: A nondeterministic planning domain $\mathcal{D} = (2^{\mathcal{F}}, s_0, Act, React, \alpha, \beta, \delta)$
 1: Let $\mathcal{F}$ be the set of fluents in domain.pddl and problem.pddl
 2: Let $s_0$ be the initial state from problem.pddl
 3: Let $Act$ and $\alpha$ be actions and preconditions from domain.pddl and problem.pddl
 4: for $s \in 2^{\mathcal{F}}$:
 5:     for $a \in Act$:
 6:         if $a \in \alpha(s)$:
 7:             Let $\{s_1, \cdots, s_n\}$ be the successor states of $s$ specified in $a$'s oneof clause
 8:             UPDATE: $React = React \cup \{r_1, \cdots, r_n\}$
 9:             DEFINE: $\delta(s, a, r_i) = s_i$ and $r_i \in \beta(s, a)$ for every $i$
10: Return $\mathcal{D} = (2^{\mathcal{F}}, s_0, Act, React, \alpha, \beta, \delta)$
```

▶ **Example 2.** Consider a robotic agent that operates in a blocksworld environment where it can pick up/drop blocks from/in top of other blocks as well as pick up/drop blocks from/on the table. The domain defines the predicates `emptyhand/0`, `holding/1`, `on-table/1`, `on/2`, `clear/1` to specify that the agent holds no block, the agent holds a block, a block is on the table, a block is on top of another block, and a block can be picked, respectively.

Assume that there exist two blocks, `yellow` and `green`, and that in the initial state `green` is on the table, `yellow` is on top of `green`, and the agent holds no block. Consider the agent actions `pick-up` and `put-on-block` defined as follows.

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

```
                       (and
                           (clear ?b2)
                           (on-table ?b1)
                           (not (on ?b1 ?b2)))))

(: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)))))
```

Intuitively, action `pick-up` specifies that whenever the agent tries to pick up a block, either it succeeds, or it does not and the block falls on the table. Similarly for action `put-on-block`. We can capture the nondeterministic effects of actions `pick-up` and `put-on-block` with two reactions `succeed` and `fail`. A fragment of the planning domain resulting from applying Algorithm 2 to the PDDL description above is shown in Figure 1.

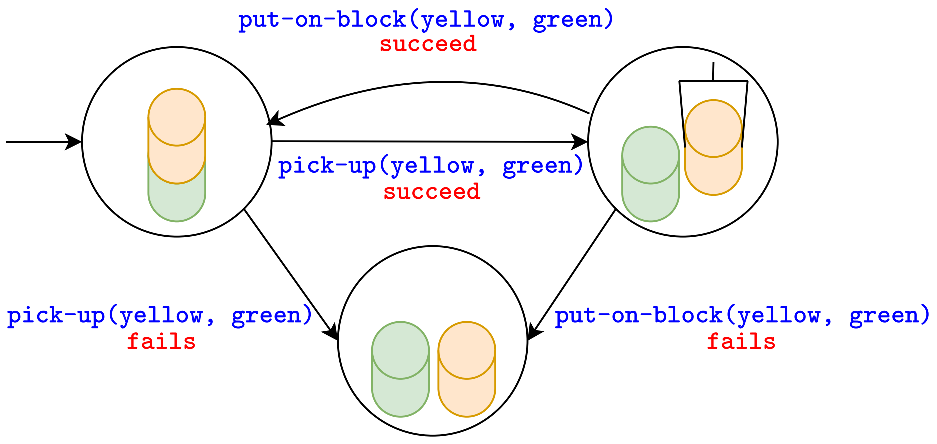

Figure 1 Fragment of the domain resulting from applying Algorithm 2 to the PDDL description in Example 2.

▶ Remark 3. Algorithm 2 returns a domain $\mathcal{D} = (2^{\mathcal{F}}, s_0, Act, React, \alpha, \beta, \delta)$ with fluents, initial state, agent actions, and action preconditions as in its PDDL description. The number of environment reactions is at most single-exponential in the number of fluents $|\mathcal{F}|$. To see this, observe that there may exist an action $a \in Act$ that, from some state $s \in 2^{\mathcal{F}}$ of the domain, nondeterministically leads to every other state $s' \in 2^{\mathcal{F}}$, thus generating an environment reaction for each such successor state. However, in practice the number of reactions is often several orders of magnitude less than $2^{\mathcal{F}}$, since the possible nondeterministic effects in `oneof` clauses of agent actions are usually very few.

Once the nondeterminism in the domain is represented through deterministic action-reaction pairs (a, r), we can construct the symbolic DFA of the domain. To do this, we characterize each pair (a, r) in terms of its add-list $\texttt{add}(a, r)$ and delete-list $\texttt{del}(a, r)$, which are the sets of fluents added and deleted by (a, r), respectively. Formally, a fluent f is in $\texttt{add}(a, r)$ (resp. $\texttt{del}(a, r)$) iff for every $s \in 2^{\mathcal{F}}$, if $a \in \alpha(s)$ and $f \notin s$ (resp. $f \in s$), then $f \in \delta(s, a, r)$ (resp. $f \notin \delta(s, a, r)$). The add- and delete-lists of action-reaction pairs can be extracted immediately from the PDDL description of the domain. Specifically, fluents appearing without (resp. with) `not` in the `oneof` clause of an action a are in the add-list (resp. delete-list) of the corresponding action-reaction pair (a, r).

We give in Algorithm 3 a technique to transform a FOND domain with a goal into a symbolic DFA. Algorithm 3 constructs a symbolic DFA with a state variable for each fluent and two error state variables, `AgErr` and `EnvErr`, denoting that the agent and the environment violated the domain specification, respectively, and whose initial state is that of the domain (Line 1). The alphabet of the symbolic DFA is partitioned into actions and reactions, which are under the control of the agent and the environment, respectively (Line 2). For every state variable corresponding to a fluent, Algorithm 3 constructs its transition function as specified in Line 3. Intuitively, the transition function η_f of fluent f specifies that in the next time step f holds if and only if either:

1. f was true in the previous time step and was not deleted by an action-reaction pair (a, r) such that $f \in \texttt{del}(a, r)$; *or*
2. f was added by some action reaction pair (a, r) such that $f \in \texttt{add}(a, r)$.

Algorithm 3 constructs the transition functions of `AgErr` and `EnvErr` in Lines 4 and 5. Intuitively, the transition function $\eta_{\texttt{AgErr}}$ of `AgErr` specifies that in the next time step the agent reaches the error state if and only if either:

1. The agent was in its error state in the previous time step, written `AgErr`; *or*
2. The agent violated the mutual exclusion axiom for its actions, which states that, at each time step, the agent must execute one and only one action, written $\neg\texttt{AgMutex}(\mathcal{Y})$;
3. The agent violated an action precondition, written $\neg\texttt{AgPre}(\mathcal{Z}, \mathcal{Y})$.

The transition function of `EnvErr` is constructed similarly.

Taking the agent's point of view, Algorithm 3 constructs the final states of the DFA so that a trace is accepted if the agent does not reach its error state and either the environment reaches its error state or the goal is reached (Line 6).

The size of the symbolic DFA constructed by Algorithm 3, i.e., the number of its state variables, is polynomial in the size of the domain. Each line can be executed in polynomial time in the size of the domain. As a result, Algorithm 3 runs in polynomial time in the size of the input domain.

Together, Algorithms 2 and 3 form our technique for transforming PDDL into symbolic DFA. While worst-case exponential in the size of the input domain, our transformation is often polynomial due to its compact representation of nondeterministic effects of agent actions with suitable environment reactions.

4 Reduction of FOND Planning to Synthesis

As an application to demonstrate the effectiveness of our PDDL to DFA transformation technique, we show how to use it to solve FOND planning problems.

Algorithm 3 DomainToDfa$(\mathcal{D}, G)$.

```
Require: A FOND domain $\mathcal{D} = (2^{\mathcal{F}}, s_0, Act, React, \alpha, \beta, \delta)$ with goal $G$
Ensure: A symbolic DFA $\mathcal{A}^s = (\mathcal{Z}, \mathcal{X}, \mathcal{Y}, Z_0, \eta, f)$
1: Define $\mathcal{Z} = \mathcal{F} \cup \{\texttt{AgErr}, \texttt{EnvErr}\}$
2: Define initial state $Z_0 = s_0$
3: Define $\mathcal{Y} = Act$
4: Define $\mathcal{X} = React$
5: for each $f \in \mathcal{F}$:
       $\eta_f(\mathcal{Z}, \mathcal{X}, \mathcal{Y}) = \left( f \wedge \neg \bigvee_{(a,r) \mid f \in \texttt{del}(a,r)} (a \wedge r) \right) \vee \bigvee_{(a,r) \mid f \in \texttt{add}(a,r)} (a \wedge r)$
6: Define agent error transition:
       $\eta_{\texttt{AgErr}}(\mathcal{Z}, \mathcal{X}, \mathcal{Y}) = \texttt{AgErr} \vee \neg\texttt{AgMutex}(\mathcal{Y}) \vee \neg\texttt{AgPre}(\mathcal{Z}, \mathcal{Y})$
       where:
           $\texttt{AgMutex}(\mathcal{Y}) = \left( \bigvee_{a \in Act} a \right) \wedge \left( \bigwedge_{a,a' \in Act, a \neq a'} a \supset \neg a' \right)$
           $\texttt{AgPre}(\mathcal{Z}, \mathcal{Y}) = \bigwedge_{a \in Act} \left( a \supset \bigvee_{s \in 2^{\mathcal{F}}, a \in \alpha(s)} s \right)$
7: Define environment error transition:
       $\eta_{\texttt{EnvErr}}(\mathcal{Z}, \mathcal{X}, \mathcal{Y}) = \texttt{EnvErr} \vee \neg\texttt{EnvMutex}(\mathcal{X}) \vee \neg\texttt{EnvPre}(\mathcal{Z}, \mathcal{Y}, \mathcal{X})$
       where:
           $\texttt{EnvMutex}(\mathcal{X}) = \left( \bigvee_{r \in React} r \right) \wedge \left( \bigwedge_{r,r' \in React, r \neq r'} r \supset \neg r' \right)$
           $\texttt{EnvPre}(\mathcal{Z}, \mathcal{Y}, \mathcal{X}) = \bigwedge_{r \in React} \left( r \supset \bigvee_{(s,a) \in (2^{\mathcal{F}} \times Act), r \in \beta(s,a)} (s \wedge a) \right)$
8: Define accepting condition:
       $f(\mathcal{Z}) = \neg\texttt{AgErr} \wedge (\texttt{EnvErr} \vee G)$
9: Return $\mathcal{A}^s = (\mathcal{X}, \mathcal{Y}, \mathcal{Z}, Z_0, \eta, f)$
```

Let $\mathcal{D}$ and G be a FOND domain and goal in PDDL. Construct its symbolic DFA $\mathcal{A}^s$ by using Algorithms 2 and 3. Solve the symbolic DFA game over $\mathcal{A}^s$, assigning actions and reactions to agent and environment, respectively.

▶ **Theorem 4.** *Let $\mathcal{D}$ and G be a FOND domain and goal in PDDL and $\mathcal{A}^s$ their DFA. There is a strong plan for G in $\mathcal{D}$ iff there is a winning strategy in $\mathcal{A}^s$.*

This synthesis technique is exponential in the size of the PDDL domain and therefore optimal wrt the computational complexity of FOND planning. However, we also observe that synthesis on DFA games is based on basic backward search algorithms. While we do not introduce sophisticated optimization techniques, we show that we can include them in our synthesis algorithm.

Specifically, we show how to include a simple form of *invariants*, i.e., properties of states of the domain that must remain unchanged during the execution of every sequence of actions [12]. Invariants can be used to prune the search space by eliminating actions or states that violate these unchanging properties.

We consider *mutual exclusion* invariants specifying that, at every state, no more than one fluent or negation of fluent l in a set I can be true. Such invariants can be captured as a Boolean formula $i(\mathcal{Z})$ over the state variables of the symbolic DFA constructed in Algorithm 3 as:

$$i(\mathcal{Z}) = \bigwedge_{l \in I} (l \supset \bigwedge_{l' \in I . l \neq l'} \neg l')$$

To include invariants in backward search, we rewrite the fixpoint computation in Section 2. Specifically, w and t are now initialized as $w_0(\mathcal{Z}) = f(\mathcal{Z}) \wedge i(\mathcal{Z})$ and $t_0(\mathcal{Z}, \mathcal{Y}) = f(\mathcal{Z}) \wedge i(\mathcal{Z})$, since every goal state must satisfy the invariant. Then, t_{i+1} and w_{i+1} are constructed as follows:

$$t_{i+1}(\mathcal{Z}, \mathcal{Y}) = t_i(\mathcal{Z}, \mathcal{Y}) \vee (\neg w_i(\mathcal{Z}) \wedge \forall \mathcal{X} . w_i(\eta(\mathcal{X}, \mathcal{Y}, \mathcal{Z})))$$

$$w_{i+1}(\mathcal{Z}) = (\exists \mathcal{Y} . t_{i+1}(\mathcal{Z}, \mathcal{Y})) \wedge i(\mathcal{Z})$$

That is, only states satisfying the invariant are added to the winning region.

► Remark 5. An approach alternative to ours is to reduce FOND planning to LTL$_f$ synthesis of the formula $\mathcal{E} \supset \varphi$, where $\mathcal{E}$ and φ describe the domain and the agent goal, respectively [2]. The LTL$_f$ formula is transformed in DFA in doubly-exponential time (Theorem 1) and a strong plan is obtained by solving the corresponding DFA game. However, this approach is limited by the doubly-exponential blowup resulting from transforming the LTL$_f$ formula in DFA.

5 Evaluation

We implemented Algorithms 2 and 3 in a tool called PDDL2DFA. For parsing, grounding, and computing invariants for the input PDDL domain, we based on the tool PRP [32]. We code Boolean functions representing transitions and final states of symbolic DFAs by Binary Decision Diagrams (BDDs) [13] with the BDD library CUDD 3.0.0 [36]. The size of a BDD is the number of its nodes. PDDL2DFA also implements the transformation from PDDL to LTL$_f$ (see Remark 5), in which case the DFAs of LTL$_f$ formulas are constructed with LYDIA [21], which is among the best performing tools publicly available for LTL$_f$-to-DFA conversion.

We applied the transformation in PDDL2DFA to implement the reduction of FOND planning to synthesis (ref. Section 4) in a tool called SYFT4FOND[2]. For solving symbolic DFA games, we use the symbolic synthesis framework in [40] at the base of state-of-the-art LTL$_f$ synthesis tools [11, 37]. We compute winning strategies for DFA games through Boolean synthesis [28].

[2] PDDL2DFA and SYFT4FOND at `https://github.com/GianmarcoDIAG/syft4fond`

Setup. Experiments were run on a laptop running 64-bit Ubuntu 22.04, 3.6 GHz CPU, and 12 GB of memory. Timeout was 1000 secs.

5.1 Benchmark

We performed experiments on a suite of 170 classical FOND planning benchmarks divided in five classes: blocks world (50 instance), extended blocks world (50 instances), triangle-tire world (40 instances), rectangle-tire world (15 instances), and elevators (15 instances). In blocks world instances, the agent manipulates blocks with actions that can nondeterministically succeed or fail. In extended blocks world instances, the agent can also move towers of two blocks, with similar nondeterministic success or failure in actions. In triangle-tire and rectangle-tire instances, the agent navigates a grid environment, dealing with nondeterministic success or failure when moving. Elevator instances involve managing elevators to collect coins across multiple floors. For each class, the instances grow by increasing the problem size, which depends on their parameters.

5.2 Empirical Results

We performed experiments to evaluate the efficiency of our technique for translating PDDL into symbolic DFA and the practical feasibility of reducing FOND planning to synthesis.

We evaluated the performance of PDDL2DFA in transforming PDDL to LTL$_f$ and LTL$_f$ to symbolic DFA. When employing this transformation, PDDL2DFA was unable to construct the symbolic DFA of the PDDL domain in the considered benchmarks, except in very few cases. The bottleneck was transforming LTL$_f$ into DFA, which requires doubly-exponential time in the size of the LTL$_f$ formula [26].

Table 1 Coverage achieved by PDDL2DFA (constructed DFAs/instances in benchmark), and domain size ($|\mathcal{F}|$), number of actions ($|Act|$), size of largest fluent BDD (Max$_{\{\mathcal{F}\}}$ BDD), size of smallest fluent BDD (Min$_{\{\mathcal{F}\}}$ BDD), size of agent error BDD (`AgErr` BDD), and size of environment error BDD (`EnvErr` BDD) in the largest solved instance.

Bench.	Coverage	$\lvert\mathcal{F}\rvert$	$\lvert Act\rvert$	Max$_{\{\mathcal{F}\}}$ BDD	Min$_{\{\mathcal{F}\}}$ BDD	`AgErr` BDD	`EnvErr` BDD
Blocks	31/50	1055	1953	514	14	6549	109
BlocksExt	19/50	379	12654	5085	277	156509	562
Triangle	30/40	2881	4709	1305	17	40247	740
Rect.	10/15	53	52249	5432	2077	66261	1783
Elev.	15/15	66	105	42	10	511	41
Total	**105**/170	–	–	–	–	–	–

We evaluated the performance of PDDL2DFA in transforming PDDL to symbolic DFA with Algorithms 2 and 3. Table 1 shows that PDDL2DFA constructed the DFA for a considerable number of benchmarks. Notably, PDDL2DFA was able to construct the DFA of triangle-tire domains with side 61. The bottleneck of the DFA construction was computing the agent error BDD. Indeed, Table 1 shows that the size of the agent error BDD in the hardest solved instances is orders of magnitude larger than that of the other BDDs, including that of the largest fluent BDD. The reason is that constructing the BDD of the agent error requires iterating over all actions of the domain. Instead, constructing the BDDs of the fluents requires only considering actions containing that fluent in their add- and delete-lists. Constructing the environment error BDD requires iterating over all environment reactions, but these are often several orders of magnitude less than fluents and agent actions (see Remark 3). As a result, the number of actions is the parameter that affects most the performance of the DFA construction. The size of the PDDL domain, i.e., its fluents, has less impact than the number of agent actions on the DFA construction performance. Overall, this is a good result and shows the effectiveness of our construction.

Table 2 Coverage achieved by SYFT4FOND (solved instances/instances in benchmark), and domain size ($|\mathcal{F}|$), number of actions ($|Act|$), size of largest fluent BDD ($\text{Max}_{\{\mathcal{F}\}}$ BDD), size of smallest fluent BDD ($\text{Min}_{\{\mathcal{F}\}}$ BDD), size of agent error BDD (`AgErr` BDD), and size of environment error BDD (`EnvErr` BDD) in the largest solved instance.

Bench.	Coverage	$\|\mathcal{F}\|$	$\|Act\|$	$\text{Max}_{\{\mathcal{F}\}}$ BDD	$\text{Min}_{\{\mathcal{F}\}}$ BDD	`AgErr` BDD	`EnvErr` BDD
Blocks	6/50	55	78	46	10	403	29
BlocksExt	3/50	19	81	33	8	327	32
Triangle	9/40	298	467	169	12	2119	111
Rect.	8/15	25	15481	2272	1057	74757	884
Elev.	7/15	44	68	28	10	281	28
Total	**33**/170	-	-	-	-	-	-

We evaluated the performance of SYFT4FOND in reducing FOND planning to synthesis. Table 2 shows that SYFT4FOND is able to solve a reasonable number of instances. Indeed, SYFT4FOND was able to solve triangle-tire planning instances with side 19, though based on plain backward search. The bottleneck of the synthesis was constructing the BDDs of the agent winning moves and region during the fixpoint computation, which are mostly affected by the size of the agent error BDD. In general, we consider this result adequate to show the practical feasibility of reducing FOND planning to synthesis.

6 Conclusion

In this paper, we have presented an effective method for translating PDDL into symbolic DFA. We implemented our method in a new tool, PDDL2DFA, and applied it to solving planning problems through reduction to synthesis in the tool SYFT4FOND. Testing these tools on various case studies demonstrated the practicality and performance of our approach. Indeed, we demonstrated that our method successfully constructs DFAs for considerably large domains and solves a practical number of planning instances, performing significantly better than straightforward approaches based on translating PDDL in LTL_f. Our work makes a step towards integrating planning and synthesis more closely. Future research can build on this foundation, aiming to integrate FOND domains into advanced synthesis techniques developed for temporally extended goals [27, 19], structured environment specifications [20, 9, 30, 3], multiple goal specifications [14, 38], and for handling goal unrealizability [4, 5, 24, 23].

References

1 Benjamin Aminof, Giuseppe De Giacomo, Aniello Murano, and Sasha Rubin. Planning and synthesis under assumptions. *CoRR*, abs/1807.06777, 2018. `arXiv:1807.06777`.

2 Benjamin Aminof, Giuseppe De Giacomo, Aniello Murano, and Sasha Rubin. Planning under LTL environment specifications. In *ICAPS*, pages 31–39, 2019. URL: `https://ojs.aaai.org/index.php/ICAPS/article/view/3457`.

3 Benjamin Aminof, Giuseppe De Giacomo, Gianmarco Parretti, and Sasha Rubin. Effective approach to ltlf best-effort synthesis in multi-tier environments. In *IJCAI*, 2024.

4 Benjamin Aminof, Giuseppe De Giacomo, and Sasha Rubin. Best-effort synthesis: Doing your best is not harder than giving up. In *IJCAI*, pages 1766–1772, 2021. doi:10.24963/IJCAI.2021/243.

5 Benjamin Aminof, Giuseppe De Giacomo, and Sasha Rubin. Reactive synthesis of dominant strategies. In *AAAI*, pages 6228–6235. AAAI Press, 2023. doi:10.1609/AAAI.V37I5.25767.

6 Benjamin Aminof, Giuseppe De Giacomo, Aniello Murano, and Sasha Rubin. Planning under LTL environment specifications. In *ICAPS*, pages 31–39. AAAI Press, 2019. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/3457.

7 Krzysztof R. Apt and Erich Grädel, editors. *Lectures in Game Theory for Computer Scientists.* Cambridge University Press, 2011.

8 Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking.* MIT Press, 2008.

9 Suguman Bansal, Giuseppe De Giacomo, Antonio Di Stasio, Yong Li, Moshe Y. Vardi, and Shufang Zhu. Compositional safety LTL synthesis. In *VSTTE*, volume 13800 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2022. doi:10.1007/978-3-031-25803-9_1.

10 Suguman Bansal, Yong Li, Lucas M. Tabajara, and Moshe Y. Vardi. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *AAAI*, pages 9766–9774, 2020. doi:10.1609/AAAI.V34I06.6528.

11 Suguman Bansal, Yong Li, Lucas M. Tabajara, and Moshe Y. Vardi. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *AAAI*, pages 9766–9774, 2020. doi:10.1609/AAAI.V34I06.6528.

12 Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997. doi:10.1016/S0004-3702(96)00047-1.

13 Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. doi:10.1145/136035.136043.

14 Alberto Camacho, Meghyn Bienvenu, and Sheila A. McIlraith. Finite LTL synthesis with environment assumptions and quality measures. In *KR*, pages 454–463. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/KR/KR18/paper/view/18072.

15 Alberto Camacho, Meghyn Bienvenu, and Sheila A McIlraith. Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, pages 58–67, 2019. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/3460.

16 Alonzo Church. Logic, arithmetic and automata. In *Proc. International Congress of Mathematicians*, 1963.

17 Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147:35–84, 2003. doi:10.1016/S0004-3702(02)00374-0.

18 Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong Planning in Non-Deterministic Domains Via Model Checking. In *AIPS*, pages 36–43. AAAI, 1998. URL: http://www.aaai.org/Library/AIPS/1998/aips98-005.php.

19 Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, and Sasha Rubin. Pure-past linear temporal and dynamic logic on finite traces. In *IJCAI*, pages 4959–4965, 2020. doi:10.24963/IJCAI.2020/690.

20 Giuseppe De Giacomo, Antonio Di Stasio, Moshe Y Vardi, and Shufang Zhu. Two-stage technique for LTL$_f$ synthesis under LTL assumptions. In *KR*, pages 304–314, 2020. doi:10.24963/KR.2020/31.

21 Giuseppe De Giacomo and Marco Favorito. Compositional approach to translate LTL$_f$/LDL$_f$ into deterministic finite automata. In *ICAPS*, pages 122–130, 2021. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/15954.

22 Giuseppe De Giacomo and Yves Lespérance. The nondeterministic situation calculus. In *KR*, pages 216–226, 2021. doi:10.24963/KR.2021/21.

23 Giuseppe De Giacomo, Gianmarco Parretti, and Shufang Zhu. LTL$_f$ best-effort synthesis in nondeterministic planning domains. In *ECAI*, pages 533–540, 2023. doi:10.3233/FAIA230313.

24 Giuseppe De Giacomo, Gianmarco Parretti, and Shufang Zhu. Symbolic LTL$_f$ best-effort synthesis. In *EUMAS*, pages 228–243, 2023.

25 Giuseppe De Giacomo and Sasha Rubin. Automata-theoretic foundations of FOND planning for LTL$_f$ and LDL$_f$ goals. In *IJCAI*, pages 4729–4735, 2018. doi:10.24963/IJCAI.2018/657.

26 Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

27 Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on finite traces. In *IJCAI*, pages 1558–1564, 2015. URL: http://ijcai.org/Abstract/15/223.

28 Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi. BDD-based Boolean functional synthesis. In *CAV*, pages 402–421, 2016. doi:10.1007/978-3-319-41540-6_22.

29 Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, 2013.

30 Giuseppe De Giacomo, Antonio Di Stasio, Lucas M. Tabajara, Moshe Y. Vardi, and Shufang Zhu. Finite-trace and generalized-reactivity specifications in temporal synthesis. *Formal Methods Syst. Des.*, 61(2):139–163, 2022. doi:10.1007/S10703-023-00413-2.

31 Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool, 2019.

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

33 Gianmarco Parretti. Syft4Fond. Software (visited on 2025-09-18). URL: https://github.com/GianmarcoDIAG/syft4fond, doi:10.4230/artifacts.24786.

34 A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

35 Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

36 Fabio Somenzi. CUDD: CU Decision Diagram Package 3.0.0. Universiy of Colorado at Boulder, 2016.

37 Shufang Zhu and Marco Favorito. Lydiasyft: A compositional symbolic synthesis framework for ltl$_\mathrm{f}$ specifications. In *TACAS 2025*, pages 295–302, 2025. doi:10.1007/978-3-031-90643-5_15.

38 Shufang Zhu and Giuseppe De Giacomo. Act for your duties but maintain your rights. In *KR*, 2022.

39 Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. Symbolic LTL$_f$ synthesis. In *IJCAI*, pages 1362–1369, 2017. doi:10.24963/IJCAI.2017/189.

40 Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. Symbolic LTL$_f$ synthesis. In *IJCAI*, pages 1362–1369, 2017. doi:10.24963/IJCAI.2017/189.