Safe-Planner: A Single-Outcome Replanner for Computing Strong Cyclic Policies in Fully Observable Non-Deterministic Domains

Vahid Mokhtari, Ajay Suresha Sathya, Nikolaos Tsiogkas, Wilm Decré

***Abstract*— Replanners are efficient methods for solving non-deterministic planning problems. Despite showing good scalability, existing replanners often fail to solve problems involving a large number of *misleading plans*, i.e., weak plans that do not lead to strong solutions, however, due to their minimal lengths, are likely to be found at every replanning iteration. The poor performance of replanners in such problems is due to their all-outcome determinization. That is, when compiling from non-deterministic to classical, they include all compiled classical operators in a single deterministic domain which leads replanners to continually generate misleading plans. We introduce an offline replanner, called Safe-Planner (SP), that relies on a single-outcome determinization to compile a non-deterministic domain into a set of classical domains, and ordering heuristics for ranking the obtained classical domains. The proposed single-outcome determinization and the heuristics allow for alternating between different classical domains. We show experimentally that this approach can allow SP to avoid generating misleading plans but to generate weak plans that directly lead to strong solutions. The experiments show that SP outperforms state-of-the-art non-deterministic solvers by solving a broader range of problems. We also validate the practical utility of SP in real-world non-deterministic robotic tasks.**

I. Introduction

AI planning is the reasoning side of acting. It is a core deliberation function for intelligent robots to increase their autonomy and flexibility through the construction of sequences of actions for achieving some prestated objectives [1]. Major progress has been achieved in developing classical AI planning algorithms, however, these approaches are restricted to the hypothesis of deterministic actions: an applicable action in a state results in a single new state, thus the world evolves along a single fully predictable path. A more realistic assumption must take into account that the effect of actions can not be completely foreseen and the environment can change in unpredictable ways, however, an agent is able to observe its outcome. As an example, let us consider a dualarm manipulator in a non-deterministic task involving picking objects from a table and putting them in a box in a certain orientation (see Figure 1). The orientation of objects on the table are not known to the robot. In order to rotate objects in a certain orientation, the robot needs to observe the objects at runtime. So, a strong solution for the robot must be contingent on observing the orientation of objects.

We address Fully Observable Non-Deterministic (FOND) planning problems as an attempt at modeling uncertainty into the outcomes of actions [2]. Solutions to FOND problems are

All authors are with the Department of Mechanical Engineering, Division Robotics, Automation and Mechatronics (RAM), and Flanders Make, KU Leuven, Belgium. Email: mokhtari.vahid@kuleuven.be

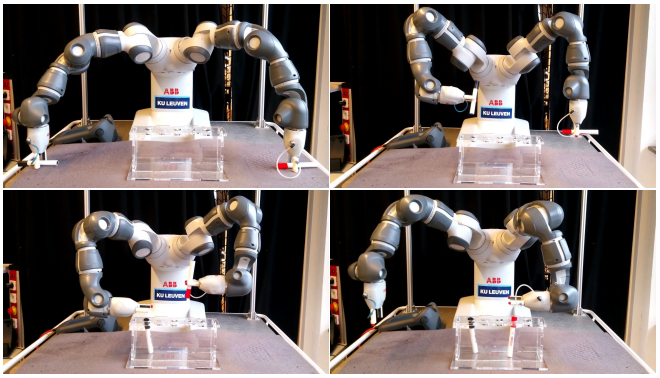


Fig. 1: A dualarm YuMi manipulator puts two objects into a box in an upward orientation. A strong solution, encoding contingencies, for the robot involves picking up objects; carrying the objects to a camera; checking the orientation of the objects; rotating the objects or not, depending on their observed orientation; and putting them into the box

policies, i.e., mappings from states to actions, that reach a goal state when executed from the initial state of problems. The execution of a policy may result in more than one sequence of states. Therefore, solutions to FOND planning problems are characterized with respect to the different possible executions of plans. In [3] three classes of solutions are presented: (*i*) *weak solutions* having a chance to achieve the goal; (*ii*) *strong solutions* guaranteed to achieve the goal; and (*iii*) *strong cyclic solutions* having a possibility to terminate and if they terminate, guaranteed to achieve the goal.

Strong cyclic solutions have become an important concept in FOND planning as they deal with *deadends*, i.e., unsolvable states where replanning is not possible. As a result, notable FOND planning techniques have been developed over the past years including planners relying on classical algorithms (referred to as replanners) such as PRP [4], FIP [5], and NDP [6], and non-classical planners relying on binary decision diagrams (BDDs) such as MBP [3], and SAT planning such as FOND-SAT [7].

Replanners compute a weak policy that does not address all non-deterministic reachable states. During the execution if an unpredicted state is reached, they compute a new policy starting from the unpredicted state. They repeat this process until every possible execution of the policy achieves the goal. Replanners have been shown to scale up best to existing FOND benchmarks from the past International Planning Competitions (IPC) [4], [5]. However, in [7], authors show that most of the FOND benchmarks from IPC do not pose a big challenge to replanners, except for large problems. They introduce domains that give rise to an exponential number of *misleading plans*, i.e., weak plans involving unsolvable states that do not lead to strong policies, however, due to their

minimal lengths, are likely to be found by replanners, and show existing replanners tend to break on these problems.

Replanners strongly rely on their adopted determinization strategy and the initial weak plan that they generate and refine toward a final strong solution. There are two known methods for compiling from non-deterministic to deterministic domains [8]: (*i*) *single-outcome* which selects and includes only one outcome for each non-deterministic operator in the deterministic domain, and (*ii*) *all-outcome* which selects and includes every outcome as a distinct operator in the deterministic domain. All-outcome replanners compile a non-deterministic domain into a single deterministic domain, leading to continually generating misleading plans (in the presence of such plans in a FOND problem). Single-outcome replanners, by contrast, compile a non-deterministic domain into a set of deterministic domains which allow to compute varying weak plans that some of them might directly lead to full strong policies.

We present Safe-Planner (SP), an offline replanner motivated by NDP2 [9], that makes the following contribution: SP adopts a single-outcome determinization to solve FOND problems involving misleading plans. SP generates a set of single-outcome deterministic domains from a non-deterministic domain and ranks them according to an ordering heuristic, e.g., on the number of planning operators' effects. In order not to lose the completeness, SP also includes all-outcome determinization in the set of compiled domains. We show experimentally this determinization strategy allows SP to compute weak plans that can directly lead to strong policies in problems involving misleading plans. SP is provably shown to be sound and complete and its code is publicly provided at: https://github.com/mokhtarivahid/safe-planner.

SP adopts a similar strategy of NDP2 in solving FOND problems, i.e., in dealing with unsolvable states and computing weak plans. SP integrates off-the-shelf classical planners for its internal reasoning without any modification. When deadends are identified during FOND planning, SP modifies a planning problem to prevent a classical planner from finding weak plans involving actions that lead to deadends. SP also adopts a *multiprocessing* approach; that is, SP simultaneously calls different solvers to make weak plans and as soon as one solver finds a plan, it terminates other solvers, thus exploiting the strength of each individual planner in solving problems, e.g., heuristic state space planners have been considered stronger on many non-optimal planning problems and SAT planners usually give optimality guarantees.

We evaluate the performance of SP in a set of existing FOND benchmarks from the past IPCs and literature. We compare our results to PRP and FOND-SAT, as well as the basic algorithm of NDP2, and show that SP outperforms these FOND solvers, particularly in the introduced domains in [7] involving a large number of misleading plans. The experiments show the adopted single-outcome determinization can have a remarkable impact on replanners' performance. We also present successful integrations of SP into simulated and real robot applications in non-deterministic tasks.

II. Related Work

FF-Replan [8] is an early replanner that uses a classical planner for reasoning. FF-Replan is a reactive online planning algorithm that generates all-outcomes determinization of a non-deterministic domain and using the classical planner FF [10] makes a weak policy to a problem. If the execution of the weak policy fails in an unexpected state in the non-deterministic environment, FF is reinvoked to make a new weak policy from the failed state. FF-Replan repeats this procedure until a goal state is reached. FF-Replan has been demonstrated to be very effective in many non-deterministic planning problems, however, it does not generate strong policies, and is hence oblivious to getting stuck in deadends.

RFF [11] and NDP [6] are two offline replanners that iteratively expand an initial plan to generate a robust policy to a problem. RFF, the winner of the probabilistic track of the 6th international planning competition (IPC2008), computes an initial weak policy by solving the all-outcome determinization of a problem using FF and then iteratively improves the robustness of the policy by finding a failed state in the non-deterministic environment and replanning from that failed state to a goal state. RFF terminates as soon as the probability to reach some failed state from the initial state is less than a threshold. RFF does not guarantee to avoid deadends, hence the generated policies are not strong solutions. A similar idea is present in NDP, however, during improving the robustness of a policy, NDP ignores any probabilities leading to deadends and thus produces strong cyclic solutions. The same group also presents NDP2 [9], an extended version of NDP with some major improvements. SP implements an NDP2 strong cyclic planning algorithm that relies on a single-outcome determinization leading to avoid generating misleading plans, and thus showing an improved performance.

FIP [5] is a customized FF-replanner which introduces some optimizations into a basic algorithm inspired by NDP. When searching for a weak plan, FIP directly removes state-action pairs that lead to deadends from the search tree, and stops the search when a solved state is revisited. FIP also introduces a goal alternative heuristic. That is, to recover from a failed state, FIP first attempts to find a plan for the intended effect of that failed state before replanning to the overall goal of the problem. Despite showing a good performance, FIP faces the same limitation on dealing with misleading plans due to its all-outcome determinization.

PRP [4] is an all-outcome replanner to compute strong cyclic plans. Instead of storing explicit state-action pairs, PRP stores a mapping of partial states to actions. A partial state is a regression from the goal to an action, containing only the relevant portion of an intended complete state. PRP detects deadends, by a fast but incomplete verification algorithm, in the delete relaxation of all-outcome deterministic planning, and computes a minimal partial state for each detected deadend state. PRP then, rather than backtracking, resets the policy and starts the computation over with the knowledge of deadend state-action pairs to restrict the ex-

pansion of such nodes. PRP shows to scale up best among the existing replanners, however it faces similar limitations on dealing with misleading plans.

FOND-SAT [7] introduces a compact SAT formulation for FOND planning. It relies on CNF encodings of atoms and non-deterministic actions of a FOND problem indexed by controller states, and running a SAT-solver over these encodings. FOND-SAT produces compact policies from a satisfying truth assignment of the problem. It performs well in problems involving many misleading plans where the problems are not too large. However, the challenge for FOND-SAT still remains at scaling up to problem size and policy size.

MBP [3] is one of the best known strong cyclic planners that formulates planning as model-checking. MBP encodes a domain description into a BDD-based representation which allows a compact representation of large sets of states. MBP then implements a wide range of algorithms, e.g., weak, strong, strong cyclic, conformant, contingent and partial observability, that could operate on top of this representation. The inherent weakness of MBP is that it does not exploit any heuristics used as in classical planning and has to search a very large state space in most planning problems, and hence does not scale up to problems of large sizes.

III. FOND Planning Problems

We present a formalism for modeling FOND planning problems which is equivalent to the PDDL representation with "oneof" clauses [12].

A *non-deterministic planning domain* is described by a pair, $D = (L, O)$, where L is a *first-order language* that has a finite number of predicate symbols and constant symbols to represent various properties of the world and O is a set of non-deterministic planning operators (as follows).

Every *non-deterministic planning operator* $o \in O$ is specified by a tuple, (h, P, E), where h is the operator's head, a functional expression of the form $n(x_1, \ldots, x_k)$ in which n is the name and $x_1, \ldots, x_k$ are the variables appearing anywhere in o, P is the precondition of o, a set of literals that must be proved in a world state in order for o to be applicable in that state, and $E = \{e_1, \ldots, e_m\}$ is a set of non-deterministic effects of o such that each e_i is a set of literals specifying the changes on a state effected by o. The effects of o are mutually exclusive, that is, only one effect happens at a time: $\mathsf{Probability}(e_i \wedge e_j) = 0$ for all pairs of integers $i, j \in \{1, \ldots, m\}$ with $i \neq j$.

The negative and positive preconditions of an operator are denoted as P^- and P^+. The negative effect, e_i^-, and the positive effect, e_i^+, are similarly defined for every $e_i \in E$.

A *state* is a set of ground predicates of L that represents all properties of the world. The set of all possible states of the world is denoted by $S \subseteq 2^{\{\text{all ground atoms of } L\}}$.

An *action* $a = (n(c_1, \ldots, c_k), P, E)$ is a ground instance of a non-deterministic operator $(n(x_1, \ldots, x_k), P, E) \in O$, where each c_i is a constant symbol of L that instantiates a variable x_i in the operator description.

In any given state s, if $P^+ \subset s$ and $P^- \cap s = \emptyset$, then a is applicable to s and the result of applying a to s is a set of states given by the following state-transition function: $\gamma(s, a) = \{(s - e_i^-) \cup e_i^+ \mid e_i \in E,\ 1 \leq i \leq m\}$.

The set of all possible actions, i.e., ground instances of planning operators in O, is denoted by A.

A fully observable *planning problem* $\mathcal{P}$ for a domain D is a tuple, $\mathcal{P} = (D, s_0, g)$, where $s_0 \in S$ is the initial state, and g is the goal, i.e., a set of propositions to be satisfied in a goal state $s_g \in S$.

A *policy* π is a set of pairs (s, a), where each $s \in S$ is a unique state in π and each $a \in A$ is an applicable action in s, denoted by $\pi(s) = \{a \mid (s, a) \in \pi,\ \gamma(s, a) \neq \emptyset\}$.

Given a policy π, the set of all states reachable by following π from s_0 is denoted by $S_\pi \subseteq S$: $S_\pi = \{s \mid (s, a) \in \pi\}$.

The *execution* of a policy π starting from a state s, denoted by $\Sigma_\pi(s)$, is a function that results in a set of terminal states:

```
1 function Σ_π(s)
2     if π = ∅ then return {s}
3     T ← {}, Q ← ⟨s⟩, V ← {}
4     while Q ≠ ∅ do
5         s ← Q.pop(), V.add(s)
6         for s' ∈ γ(s, π(s)) s.t. s' ∉ V do
7             if s' ∈ S_π then Q.add(s')
8             else T.add(s')
9     return T
```

A policy π is a *solution* for a problem $\mathcal{P} = (D, s_0, g)$ if: $|\Sigma_\pi(s_0)| > 0$, that is, there are no *inescapable* cycles in π which result in zero terminal states; and the execution of π eventually terminates in a goal state s_g such that $g \subseteq s_g$:

- π is *weak*, if $\exists s \in \Sigma_\pi(s_0) : g \not\subset s$;
- π is *strong*, if $\forall s \in \Sigma_\pi(s_0) : g \subseteq s$.

A strong policy π is *acyclic* if the execution of π never visits the same state twice; and *cyclic* otherwise, however guaranteed to eventually reach the goal in every fair execution of π [3].

IV. Classical Planning Problems

Classical planning domains are simplified models of non-deterministic planning domains where the state-transition function γ returns only one possible state per action: $|\gamma(s, a)| = 1,\ \forall s \in S,\ \forall a \in A$. Simplifying the notation and definitions given above for FOND problems: a planning operator $o = (h, P, E)$ is *classical* if $|E| = 1$, i.e., E is the single effect of o. Thus, the result of applying an action a in a state s is a new single state given by $\gamma(s, a) = (s - E^-) \cup E^+$.

Intuitively, a domain $D = (L, O)$ is classical if O is a set of classical operators, and a problem $\mathcal{P} = (D, s_0, g)$ is classical if D is a classical domain.

Solutions to classical planning problems are conventionally sequential plans rather than policies. Let $\mathcal{P} = (D, s_0, g)$ be a classical planning problem. A plan p is any sequence of actions $\langle a_1, \ldots, a_k \rangle$ for $k \geq 0$, such that there exists a sequence of states $\langle s_0, \ldots, s_k \rangle$ where each $s_i = \gamma(s_{i-1}, a_i)$ for $1 \leq i \leq k$. The plan p is a solution to $\mathcal{P}$ if $g \subseteq s_k$.

Definition 1 (Acyclic Policy Image). Let $\mathcal{P} = (D, s_0, g)$ be a classical planning problem and $p = \langle a_1, \ldots, a_k \rangle$ an *acyclic* plan for $\mathcal{P}$. The plan p can be translated into an

Algorithm 1: Compilation from FOND to Classical

Input: $D = (L, O)$ ▷ a non-deterministic domain
Output: Δ ▷ an ordered set of classical domains

1 $\Delta \leftarrow \{\}$
2 $E^n = E_1 \times \cdots \times E_n = \{(e_1, \ldots, e_n) | e_i \in E_i, 1 \leq i \leq n\}$
3 **for** $(e_1, \ldots, e_n) \in$ sorted(E^n) **do**
4 $\quad \overline{O} \leftarrow \{\}$ ▷ a set of classical operators
5 $\quad$ **for** $o_i \in O$ **do** ▷ $i \in \{1, \ldots, n\}$
6 $\quad\quad \overline{O} \leftarrow \overline{O} \cup (o_i.h, o_i.P, e_i)$
7 $\quad \Delta \leftarrow \Delta \cup (L, \overline{O})$
8 **return** $\Delta \cup (L, \bigcup_{o \in O} \bigcup_{e \in o.E} (o.h, o.P, e))$

equivalent policy $\pi = \{(s_0, a_1), \ldots, (s_{k-1}, a_k)\}$, called the *acyclic policy image* of p for $\mathcal{P}$, such that both p and π produce exactly the same sequence of state transitions at s_0, and each $s_i = \gamma(s_{i-1}, a_i)$ for $1 \leq i \leq k$. ■

V. Compilation from FOND to Classical

Let $o = (h, P, \{e_1, \ldots, e_m\})$ be a non-deterministic operator. The classical *compilation* of o is a set of classical operators $\{o_1, \ldots, o_m\}$ such that each $o_i = (h, P, e_i)$ for $1 \leq i \leq m$. We employ a single-outcome compilation to translate from non-deterministic to classical, and generate a set of all possible classical domains from the combination of the non-deterministic planning operators' outcomes.

Algorithm 1 shows our method of compiling a non-deterministic domain into a set of classical domains. It takes as input a non-deterministic domain $D = (L, O)$ including n non-deterministic operators, and compiles it into an ordered set of classical planning domains, Δ. We first build a set of all possible combinations of the non-deterministic planning operators' outcomes (line 2). We employ the n-ary Cartesian product over the n sets of the non-deterministic effects of the planning operators in O. Then, for every ordered tuple of effects, $(e_1, \ldots, e_n)$, we create and add a new classical domain to Δ by replacing the effects of every non-deterministic planning operator with every e_i (lines 3-7).

In [13], it is shown that the performance of classical planners is affected by domain model configuration. In Algorithm 1, the obtained classical domains are also ranked according to the quality of non-deterministic operators, e.g., operators with a larger number of (or fewer) effects are given the highest priority. We note there are problems where this heuristic might not be helpful in avoiding misleading plans, however, in practice, it works well on most of the benchmark problems. In this work, we do not investigate the impact of other possible heuristics for reordering planning operators. A detailed study of different heuristics has been conducted in [13], [14], which can be used and integrated in our planner.

It is worth noting that in practice the single-outcome compilation was sufficient to solve existing FOND benchmarks, however, there are situations where the single-outcome compilation can lead to incompleteness. For example, let us consider a planning task with two variables x and y, both of which are initially false. The goal is to make two variables, x and y, true. Furthermore, there is only one action a with two possible effects: either x is made true or y is made true.

Algorithm 2: Safe-Planner (SP)

Input: $(\mathcal{P} = (D, s_0, g), X)$ ▷ X is a set of classical planners
Output: π ▷ a strong cyclic solution

1 $\Delta \leftarrow$ Compilation(D) ▷ see Alg. 1
2 $\pi \leftarrow \{\}, \ \theta \leftarrow \{\}$ ▷ a set of deadend states
3 **while** $\exists s \in \Sigma_\pi(s_0)$ s.t. $g \not\subset s$ **do**
4 $\quad \pi' \leftarrow$ Make-Safe-Plan$(X, D, \Delta, s, g, \theta)$ ▷ see Alg. 4
5 $\quad$ **if** $\pi' \neq$ failure **then**
6 $\quad\quad \pi \leftarrow \pi \cup \{(s', a) \mid (s', a') \in \pi', \ a$ in D corresp. $a'\}$
7 $\quad$ **else**
8 $\quad\quad$ **if** $s = s_0$ **then return** failure ▷ no plan exists at s_0
9 $\quad\quad \theta \leftarrow \theta \cup \{s\}$ ▷ s is a deadend state
10 $\quad\quad$ **for** $(s', a') \in \pi$ s.t. $s \in \gamma_D(s', a')$ **do**
11 $\quad\quad\quad \pi \leftarrow \pi \setminus \{(s', a')\}$ ▷ remove (s', a') from π
12 **return** π

There is a strong cyclic solution, namely the application of the action a in the initial state, that leads into two states in which either x or y is true. The single-outcome yields two planning domains, a domain with a single action $\langle \text{true}, x \rangle$ and a domain with a single action $\langle \text{true}, y \rangle$. Since there is only one relevant action in each domain, both domains lead into unsolvable tasks. So, in order not to lose the completeness, we also append all-outcome determinization at the end of the list of compiled domains (line 8).

VI. Planning for Strong Cyclic Solutions

We present a non-deterministic planner, Safe-Planner (SP), for computing strong cyclic policies to FOND problems. SP is a variant implementation of NDP2 [9] that relies on a single-outcome determinization. We first describe the main algorithm of SP (Algorithm 2) which compiles a planning domain from non-deterministic to classical and aggregates solutions to classical problems and forms a strong policy. Then, we elaborate on the internal functions of SP to avoid deadends and compute acyclic plans (Algorithms 3 and 4).

In Algorithm 2, SP takes as input a FOND problem $\mathcal{P}$ including a non-deterministic domain D, an initial state s_0, and a goal g, and a set of external classical planners X, and as output computes a strong (possibly cyclic) policy π for $\mathcal{P}$.

On line 1, SP compiles D into a set of classical domains Δ (see Algorithm 1). On line 2, π is the current evolved strong policy, and θ retains a set of deadend (unsolvable) states.

On each planning iteration (lines 3-11), on line 3, SP finds a non-goal terminal state s by the execution of the current policy π on the initial state: $\Sigma_\pi(s_0)$. Note that when π is empty (e.g., in the first iteration) s is the initial state.

On line 4, Make-Safe-Plan (explained later in Algorithm 4) is a function that given the tuple $(X, D, \Delta, s, g, \theta)$ computes an acyclic policy image π' at s (if any) which never produces states in θ nor visits the same state twice.

If there is such a policy image π' at s (line 5), then it is merged into π with forcing to replace the old states in π with new ones in π' (line 6), that is, if there is a common state s' in π and π', old actions in $\pi(s')$ are replaced by new actions in $\pi'(s')$. This is the key to avoiding inescapable loops in π whereby neither a goal nor a non-goal terminal state could be found on the next iterations of SP. Note that when merging

Algorithm 3: Constrain a Domain and a Problem

Input: (D, s, A) ▷ A is a set of actions leading to deadends at s
Output: (D', s') ▷ (a constrained domain, a constrained state)

1 $D' \leftarrow D$, $s' \leftarrow s$, $(L', O') \leftarrow D'$
2 **foreach** action $n(c_1, \ldots, c_k) \in A$ **do**
3 $\quad s' \leftarrow s' \cup \{\mathsf{disallowed}_n(c_1, \ldots, c_k)\}$
4 $\quad L' \leftarrow L' \cup \{\mathsf{disallowed}_n(x_1, \ldots, x_k)\}$
5 **foreach** operator $(n(x_1, \ldots, x_k), P, E) \in O'$ **do**
6 $\quad$ **foreach** action $n(c_1, \ldots, c_k) \in A$ **do**
7 $\quad\quad P \leftarrow P \cup \{\neg\mathsf{disallowed}_n(x_1, \ldots, x_k)\}$
8 $\quad$ **foreach** action $m(c_1, \ldots, c_l) \in A$ **do**
9 $\quad\quad E \leftarrow E \cup \{\neg\mathsf{disallowed}_m(c_1, \ldots, c_l)\}$
10 **return** (D', s')

π' into π, the deterministic actions in π' are replaced with their corresponding non-deterministic actions in D (line 6).

If no such policy image exists at s (line 7), and s is the initial state (line 8), SP returns a failure; otherwise SP concludes that s is a deadend state and adds it to θ (line 9). SP also removes from π all states and actions in π that produce s (lines 10-11).

SP repeats this procedure until a strong cyclic solution is computed (lines 12) and there are no more non-goal terminal states in $\Sigma_\pi(s_0)$ (line 3); or all deadend states are detected and removed from π and no more plan exists at s_0 that avoids deadends, where planning ends in failure (line 8).

A. Constraining a Domain and a Problem

The core of SP is to avoid deadends (and cycles) when computing an acyclic policy image at a non-goal terminal state (shown later in Algorithm 4). SP uses classical planners as a black box and does not modify the internal algorithms of the planners. Therefore SP must deal with deadends externally. An approach to avoiding deadends is to look for alternative plans that do not start with already known actions leading to deadends. A method for modifying a classical planning domain and a planning problem is proposed in [9] such that actions leading to deadends become inapplicable at the first step of any solution to the problem. Based on that work, we present the Constrain function in Algorithm 3. It receives as input a classical domain D, a state s and a set of uninteresting actions A that lead to deadends (or cycles) when applied in s, and returns a constrained domain D' and a constrained state s' that ensure a classical planner cannot produce a plan starting with an action in A (if any).

On line 1, D and s are preserved unmodified by making a copy of them into D' and s'. For every action $n(c_1, \ldots, c_k)$ in A, a ground predicate $\mathsf{disallowed}_n(c_1, \ldots, c_k)$ is introduced into s' (lines 2 and 3), and an ungrounded predicate $\mathsf{disallowed}_n(x_1, \ldots, x_k)$ is introduced into L' (line 4).

For every planning operator $(n(x_1, \ldots, x_k), P, E) \in O'$ (line 5) that has instances in A (line 6), a negated ungrounded predicate $\neg\mathsf{disallowed}_n(x_1, \ldots, x_k)$ is introduced into the operator's precondition P (line 7), and for all actions $m(c_1, \ldots, c_l) \in A$ (line 8), a negated ground predicate $\neg\mathsf{disallowed}_m(c_1, \ldots, c_l)$ is introduced into the operator's effect E (line 9).

Algorithm 4: Make-Safe-Plan (MSP)

Input: $(X, D, \Delta, s_0, g, \theta)$
Output: π ▷ an acyclic policy image

1 $\pi \leftarrow \langle\rangle$; $B \leftarrow \{\}$; $U \leftarrow \theta$; $s \leftarrow s_0$
2 **loop**
3 $\quad$ **for** $\overline{D} \in \Delta$ **do**
4 $\quad\quad \overline{D'}, s' \leftarrow$ Constrain$(\overline{D}, s, \{a | (s, a) \in B\})$ ▷ Alg. 3
5 $\quad\quad$ **if** $\exists\langle \overline{a'}_1, \ldots, \overline{a'}_k\rangle$ from Planner$(X, \overline{D'}, s', g)$ **then**
6 $\quad\quad\quad$ **for** $i = 1, \ldots, k$ **do**
7 $\quad\quad\quad\quad \overline{a}_i \leftarrow$ the action in $\overline{D}$ corresp. $\overline{a'}_i$
8 $\quad\quad\quad\quad a_i \leftarrow$ the action in D corresp. $\overline{a}_i$
9 $\quad\quad\quad\quad$ **if** $\gamma_{\overline{D}}(s, \overline{a}_i) \in S_\pi \vee \gamma_D(s, a_i) \cap U \neq \emptyset$ **then**
10 $\quad\quad\quad\quad\quad B \leftarrow B \cup \{(s, \overline{a}_i)\}$
11 $\quad\quad\quad\quad\quad$ **break**
12 $\quad\quad\quad\quad \pi \leftarrow \pi \cdot (s, \overline{a}_i)$
13 $\quad\quad\quad\quad s \leftarrow \gamma_{\overline{D}}(s, \overline{a}_i)$
14 $\quad\quad\quad$ **else** ▷ inner for-loop finished with no break
15 $\quad\quad\quad\quad$ **return** π
16 $\quad\quad\quad$ **break** ▷ inner for-loop was broken, so break the outer too
17 $\quad$ **else** ▷ outer for-loop finished with no solution found for all domains
18 $\quad\quad$ **if** $\pi = \langle\rangle$ **then return** failure
19 $\quad\quad U \leftarrow U \cup \{s\}$
20 $\quad\quad (s, a) \leftarrow \pi.pop()$
21 $\quad\quad B \leftarrow B \cup \{(s, a)\}$

This ensures a grounding of an operator $n(x_1, \ldots, x_k)$ for constants $(c_1, \ldots, c_k)$ never occurs in s' containing the predicate $\mathsf{disallowed}_n(c_1, \ldots, c_k)$, as well as all predicates $\mathsf{disallowed}_m(c_1, \ldots, c_l)$ are immediately removed from s' when the first action in D' is applied in s'.

B. Computing an Acyclic Policy Image

In Algorithm 4, Make-Safe-Plan (MSP) is a function that given a tuple of a set of classical planners X, a non-deterministic domain D, a set of classical domains Δ, an initial state s_0, a goal g, and a set of deadend states θ, and using the above Constrain function, computes an acyclic policy image π at s_0 (if any) that never produces states in θ nor has cycles.

On line 1, π is the current acyclic policy image, B is a set of state-action pairs leading to cycles or deadends which no plan can begin with, U is a set of deadend states which cannot take part in any solution, and s is the current state.

Within an infinite loop (line 2), MSP iterates over the compiled classical domains in Δ (lines 3-21). On each iteration, on line 4, MSP constrains $\overline{D}$ and s into $\overline{D'}$ and s' such that no plan can be found that begins with actions associated with s in B (see Algorithm 3). On line 5, Planner is a *multiprocessing* function that calls simultaneously all external planners in X to find a plan given the constrained domain $\overline{D'}$ and the constrained state s'. As soon as a plan $\langle \overline{a'}_1, \ldots, \overline{a'}_k\rangle$ is found by a classical planner, Planner terminates other classical planners and returns the plan (line 5), and then MSP proceeds to merge the whole or part of the plan into π that avoids producing deadend states in U or causing cycles in π (lines 6-13):

Lines 6-11: for each action $\overline{a'}_i$, for $1 \leq i \leq k$ (line 6), $\overline{a}_i$ is the deterministic action in $\overline{D}$ that corresponds to $\overline{a'}_i$ (line 7) and a_i is the non-deterministic action in D that corresponds to $\overline{a}_i$ (line 8). On line 9, if the new state produced by s and $\overline{a}_i$: $\gamma_{\overline{D}}(s, \overline{a}_i)$, is in S_π (causing a cycle), or the states produced

by s and a_i: $\gamma_D(s, a_i)$, are in U (leading to deadends); then MSP stops merging the plan into π, adds the current state and action $(s, \overline{a}_i)$ in B (line 10), and breaks both for-loops (lines 11 and 16) and starts over the planning from s on line 3.

Lines 12-13: if $\overline{a}_i$ does not cause a cycle and a_i does not lead into deadends, MSP appends $(s, \overline{a}_i)$ to π (line 12), and replaces s with the new state (line 13), and continues testing and merging other actions into π (lines 6-13).

On line 15, if all actions of the current plan are merged into π, that is, no action leads into deadends nor causes cycles, MSP terminates and returns the acyclic policy image π.

On line 17, if no plan exists at s for all classical domains in Δ, MSP backtracks from s to a previous state and attempts to make a new plan while avoiding s: MSP adds s to U as an unsolvable state (line 19), and retrieves the previous state and action leading to s and adds them to B (lines 20-21). On line 18, if π is empty and there is no previous state, then this means there is no plan at s_0 that can avoid cycles nor states in U, so MSP returns failure.

Note: lines 14 and 17 conform to the for-else Python syntax, meaning the associated for-loops finished without break.

C. Soundness and Completeness

Lemma 1. *If planners in X are sound and guaranteed to terminate, then MSP is sound and returns either an acyclic policy image that never leads to deadends or a failure.*

Proof. MSP returns an acyclic policy image (Alg. 4, line 15) if it passes the test for all actions of the current plan against deadends and cycles, and since the planners in X are sound, the last action achieves the goal. Now, it is enough to show that the current plan never causes cycles nor leads to deadends. By induction, if the policy is empty, then the lemma is trivially true. We assume $\langle a_1, \ldots, a_{i-1} \rangle$ is part of the plan, already processed and merged into π, that avoids states in U and does not cause any cycle (lines 6-13). Further assume MSP is adding a new action a_i to π. On line 9, MSP already checks that a_i does not lead into deadends nor cause a cycle, and if not, a_i is added in, so $\langle a_1, \ldots, a_i \rangle$ is safe. □

Lemma 2. *If planners in X are sound, then after every iteration of SP there are no inescapable cycles in π.*

Proof. Proof by induction. By Lemma 1, the returned policy image by MSP, to merge in π, is acyclic, so when π is empty, the lemma is trivially true. For the induction step, (*i*) MSP returns an acyclic policy image, π', so SP merges and replaces old states in π with new ones in π' until the end of π', which by Lemma 1 is a goal state. So, any modified states in π must have a path to a goal; (*ii*) MSP returns a failure on a state s, so all states and actions leading to s are removed from π, and since s was the last non-goal terminal state, any state leading to s now becomes a non-goal terminal state. □

Theorem 1. *If planners in X are sound, then SP is sound.*

Proof. By Lemma 2, after every iteration of SP, there are no inescapable cycles in π. That is, all terminal states in π are reachable from s_0. If SP returns π, which means SP terminates without a failure, then there are no more non-goal terminal states in π, so all states in π have a path to a goal state, and thus π is a valid strong policy. □

Theorem 2. *If planners in X are sound and complete, then SP is complete.*

Proof. Proof by contradiction. Let us assume SP is not complete and there are a problem $\mathcal{P}$ and a strong policy π for $\mathcal{P}$, such that SP returns a failure to $\mathcal{P}$. This means MSP cannot find a plan at s_0 that avoids states in θ, so π has paths from s_0 to the goal which involve some states in θ. This is a contradiction with Lemma 1 that MSP never returns a plan including deadend states in θ. □

VII. Implementation and Empirical Evaluation

Safe-Planner has been implemented in Python and integrates several off-the-shelf classical planners. [1] We present the performance of SP in existing FOND benchmarks and validate its utility in real and simulated robotic tasks.

A. Standard FOND Benchmarks

We show the performance of SP, using Fast-Downward (FD) [15] as its internal planner, in FOND benchmarks from the past uncertainty tracks of the 5th (IPC2006) and the 6th (IPC2008) international planning competitions [12], [16], and existing FOND domains from literature [4], [7]. We performed all experiments on a machine 1.9GHz Intel Core i7 with 16GB memory with the limited planning runtime to 30 minutes. We compare our results to PRP [2] and FOND-SAT [3] with their option for computing strong cyclic policies. Unfortunately, NDP2 is not publicly available, however, by disabling the option for the single-outcome determinization in SP, we can achieve the basic algorithm of NDP2. So, we also report the performance of SP as NDP2 using the same planner FD. Furthermore, we report the results obtained by other classical planners integrated into SP, e.g., Fast-Forward (FF) [10] and Madagascar (M) [17], and in multiprocessing.

As mentioned earlier in Algorithm 1, SP ranks the compiled single-outcome domains according to the number of effects of operators. We ran SP with the best ordering heuristic, selected on a domain-by-domain basis by alternating between descending (the default ordering heuristic in SP) and ascending orders. In Table I, we report these heuristics as $\downarrow$ for descending and $\uparrow$ for ascending orders.

Table I shows the problem solving coverage for SP, NDP2, PRP and FOND-SAT in the used FOND domains. The best coverage is shown in bold. The domains are shown in two fragments. For the domains in the upper part, which are mostly from the past IPCs and do not involve many misleading plans, PRP showed an excellent coverage, SP and NDP2 showed fairly similar performance, and FOND-SAT showed a poor performance. The last 4 FOND domains, introduced in [7] where FOND-SAT shows fairly good results, involve

[1] The code is at: https://github.com/mokhtarivahid/safe-planner
[2] https://github.com/qumulab/planner-for-relevant-policies
[3] https://github.com/tomsons22/FOND-SAT

Domain (#problems)	SP_{FD}	$NDP2_{FD}$	PRP	FOND-SAT
acrobatics (8)	**8** (0) ↓	6 (0)	**8** (0)	0 (0)
beam-walk (11)	8 (0) ↓	8 (0)	**11** (0)	0 (0)
blocksworld (30)	20 (0) ↓	20 (0)	**30** (0)	0 (0)
doors (15)	10 (0) ↑	10 (0)	**12** (3)	11 (0)
elevators (15)	**15** (0) ↓	**15** (0)	**15** (0)	5 (0)
ex-blocksworld (15)	**6** (6) ↓	**6** (6)	**6** (9)	5 (0)
faults (49)	49 (0) ↑	49 (0)	**55** (0)	0 (0)
first-responders (100)	58 (25) ↓	58 (25)	**75** (25)	0 (0)
tireworld (15)	**12** (3) ↓	**12** (3)	**12** (3)	**12** (0)
triangle-tire (40)	3 (0) ↓	3 (0)	**40** (0)	2 (0)
zenotravel (15)	**15** (0) ↓	**15** (0)	**15** (0)	5 (0)
islands (60)	**60** (0) ↑	9 (0)	31 (0)	**60** (0)
miner (50)	**50** (0) ↑	16 (0)	14 (0)	44 (0)
tireworld-spiky (11)	**11** (0) ↓	**11** (0)	1 (0)	9 (0)
tireworld-truck (74)	**73** (0) ↓	23 (0)	21 (0)	67 (0)
total (514)	**398** (34)	289 (34)	346 (60)	220 (0)

TABLE I: Total number of problems for which SP, NDP2, PRP and FOND-SAT compute strong cyclic solutions. The numbers in brackets are those problems for which it is proven that no strong solutions exist within the 30 minutes. The arrows ↓ and ↑ indicate respectively the descending and ascending orderings used for ranking the compiled single-outcome domains in SP. Best coverages are shown in bold.

many misleading plans. Due to the used single-outcome compilation, SP could avoid producing misleading plans, hence showing better results. PRP and NDP2, on the other hand, showed a poor performance in these domains because of using only all-outcome determinization.

For an illustration of the relative performance of different planners, the cactus diagram in Figure 2 shows the number of problem instances that planners solved as the timeout limit is increased. Here, we also ran SP with other integrated classical planners, e.g., FF and FF+M (multiprocessing). In most of the domains, SP could achieve the best performance using either FD or FF, however, in some domains, e.g., faults and first-responders, SP showed better results in multiprocessing mode. We note that the performance of SP is degraded slightly using multiple planners due to the multiprocessing overhead, e.g., in beam-walk, SP showed slightly poor performance in multiprocessing.

B. Real World Tasks

We also evaluated and integrated SP into real world non-deterministic tasks.

1) Real Robot Object Manipulation: We set up a packaging task including an automated workspace with a dualarm YuMi manipulator and a table containing a set of objects that need to be put upward into a box. The orientation of objects on the table are not known to the robot. A non-deterministic perception action, check_orientation, identifies the orientation of each object as either upward or downward. Given a problem, SP computes a strong policy that gives the robot a choice of rotating an object or not at runtime. Snapshots of a YuMi manipulator putting two objects into a box were shown in Figure 1. In this experiment, we used OPTIC [18], a temporal planner integrated into SP, to generate a partially ordered strong policy to this task. We implemented and executed robot's skills using eTaSL [19]. See a video of this demonstration at: https://youtu.be/MciY5Gb3c7o.

2) Simulated Environments: We integrated SP into a Task and Motion Planning (TAMP) framework [20] and demonstrated this framework in two simulated non-deterministic

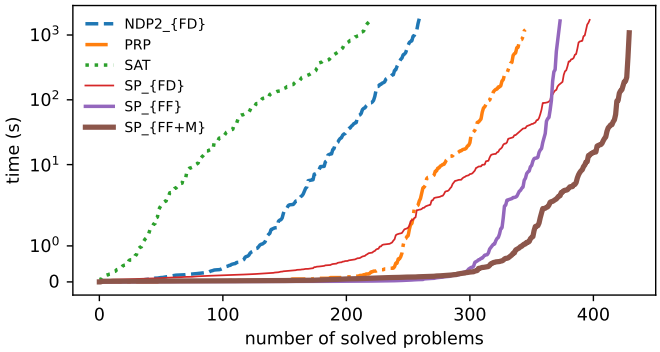
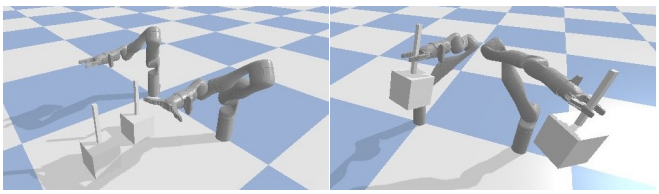


Fig. 2: Total number of solved problems as the time is increased.



(a) (b)

Fig. 3: The task is to lift the farthest object. The object might be heavy, thus the manipulator has to pull it to improve the chance of a successful lift, however, the obstructing object has to be taken away first. (a) the initial state; (b) the final configuration of the system after executing the final strong policy.

tasks. We developed a planner-independent interface which translates between the high-level symbolic task descriptions and the low-level geometrical space. In this work, task planning is achieved using SP and motion planning is achieved by solving an Optimal Control Problem (OCP) [21]. More details of this work are given in [20]. A video of these tasks is also available at: https://youtu.be/j0Ol8lKhuKk.

a) Singlearm Tabletop Object Manipulation: The first task involves lifting an object where the weight of the object is not known to the robot (see Figure 3). When the object is heavy the lift operation will fail due to torque limits, so the robot must first pull the object to a configuration that improves the robot's chance of a successful lift. We also consider some object that might obstruct the path of the pulling object. Therefore, to achieve the task, manipulators must first remove obstructing objects and then pull and lift the target object. In this experiment, the TAMP framework tests the feasibility of 11 actions, identifies 2 unfeasible actions, and makes 2 replannings in about 25 seconds. Figure 4 also shows the final strong cyclic solution, computed by SP, to achieve this task.

b) Dualarm Tabletop Object Manipulation: The second task involves picking up an object where its grasp poses might be obstructed by neighboring objects (see Figure 5). The domain allows for picking up an object with both single and dualarm actions, however, a strong solution to pick up a large object must involve two arms rather than a single arm since the object might fall down and slip from the table, resulting in a deadend state. Moreover, the obstruction constraints are not known in advance by the robot. In this experiment, the TAMP framework computed a feasible policy to achieve the task after testing the feasibility of 24 actions, identifying 9 unfeasible actions (constraints),

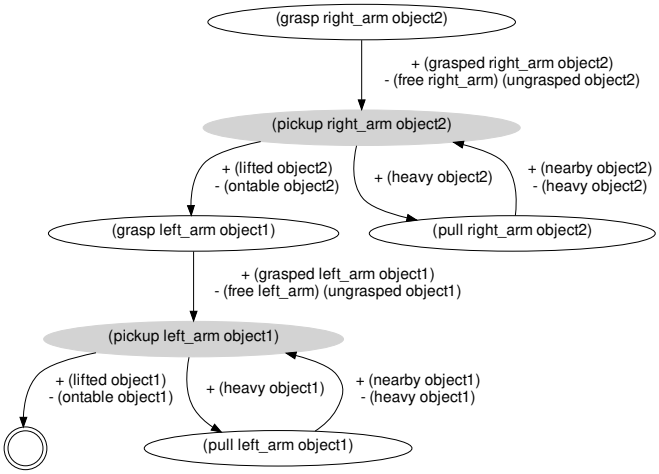


Fig. 4: A strong cyclic policy to lift an object, computed by SP. Despite involving loops, it is presumed that the plan eventually reaches a goal state under a *fairness* assumption, that is, execution cannot loop forever and outcomes leading to the goal must happen eventually.

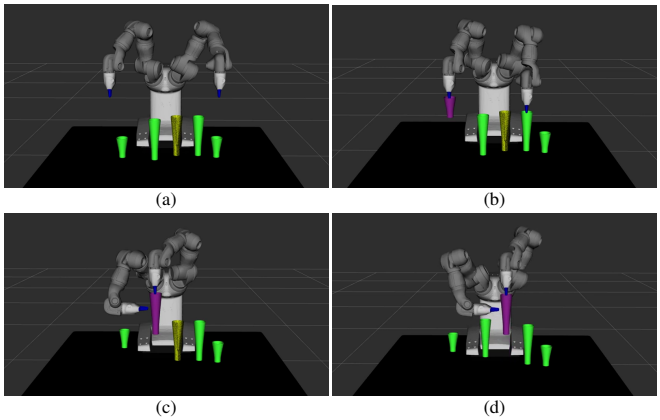


Fig. 5: Snapshots of performing a dualarm tabletop object task. (a) the initial state; (b) and (c) removing obstructing objects from the table to unblock the left grasp pose of the target object, after 9 replanning and feasibility testing; (d) the task is achieved.

and making 9 replannings in under 25 seconds.

VIII. Conclusion

We presented SP, a powerful replanner that relies on a single-outcome determinization. Using an ordering heuristic on single-outcome domains, SP produce weak plans that could directly lead onto full strong policies. SP adopts the basic ideas of NDP2 for FOND planning without any state abstraction technique. We think the integration of the single-outcome determinization with more advanced techniques such as the state relevance in PRP can result in considerable improvements in FOND planning.

The future work includes adopting different methods for configuring and reordering single-outcome domains, e.g., one similar to [22]. Other improvement includes extending SP to solve Partially Observable Non-Deterministic (POND) problems and problems with multiple initial states.

ACKNOWLEDGMENT

This work is supported by the Flanders Make SBO MULTIROB project at KU Leuven, Belgium.

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice*. Elsevier, 2004.

[2] M. Daniele, P. Traverso, and M. Y. Vardi, "Strong cyclic planning revisited," in *European Conference on Planning*, 1999, pp. 35–48.

[3] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "Weak, strong, and strong cyclic planning via symbolic model checking," *Artificial Intelligence*, vol. 147, no. 1-2, pp. 35–84, 2003.

[4] C. J. Muise, S. A. McIlraith, and C. Beck, "Improved non-deterministic planning by exploiting state relevance," in *the 22nd International Conference on Automated Planning and Scheduling*, ser. ICAPS'12. AAAI Press, 2012, pp. 172–180.

[5] J. Fu, V. Ng, F. Bastani, and I.-L. Yen, "Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems," in *the 22nd International Joint Conference on Artificial Intelligence*, ser. IJCAI'11. AAAI Press, 2011, pp. 1949–1954.

[6] U. Kuter, D. Nau, E. Reisner, and R. P. Goldman, "Using classical planners to solve nondeterministic planning problems," in *the Eighteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'08. AAAI Press, 2008, pp. 190–197.

[7] T. Geffner and H. Geffner, "Compact policies for fully observable non-deterministic planning as SAT," in *the 28th International Conference on Automated Planning and Scheduling*, ser. ICAPS'18. AAAI Press, 2018, pp. 88–96.

[8] S. W. Yoon, A. Fern, and R. Givan, "FF-Replan: A baseline for probabilistic planning," in *the Seventeenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'07, vol. 7, 2007, pp. 352–359.

[9] R. Alford, U. Kuter, D. Nau, and R. P. Goldman, "Plan aggregation for strong cyclic planning in nondeterministic domains," *Artificial Intelligence*, vol. 216, pp. 206–232, 2014.

[10] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[11] F. Teichteil-Koenigsbuch, G. Infantes, and U. Kuter, "RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure," *Sixth International Planning Competition at ICAPS*, vol. 8, 2008.

[12] B. Bonet and R. Givan, "5th international planning competition: Non-deterministic track–call for participation," 2005, the 16th International Conference on Automated Planning and Scheduling (ICAPS'06).

[13] M. Vallati, F. Hutter, L. Chrpa, and T. L. McCluskey, "On the effective configuration of planning domain models," in *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI press, 2015.

[14] M. Vallati, L. Chrpa, and T. L. McCluskey, "Improving a planner's performance through online heuristic configuration of domain models," in *The 10th Annual Symposium on Combinatorial Search*, 2017.

[15] M. Helmert, "The Fast Downward planning system," *Journal of Artificial Intelligence Research (JAIR)*, vol. 26, pp. 191–246, 2006.

[16] D. Bryce and O. Buffet, "6th international planning competition: Uncertainty part," *the 6th International Planning Competition (IPC'08)*, 2008.

[17] J. Rintanen, "Planning as satisfiability: heuristics," *Artificial Intelligence*, vol. 193, pp. 45 – 86, 2012.

[18] J. Benton, A. Coles, and A. Coles, "Temporal planning with preferences and time-dependent continuous costs," in *the 22nd International Conference on Automated Planning and Scheduling*, ser. ICAPS'12. AAAI Press, 2012, pp. 1–10.

[19] E. Aertbeliën and J. De Schutter, "etasl/etc: A constraint-based task specification language and robot controller using expression graphs," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1540–1546.

[20] A. Sathya, V. Mokhtari, W. Decre, and J. De Schutter, "Task and motion planning in fully observable non-deterministic domains," in *5th Workshop on Learning (in) Task and Motion Planning at Robotics: Science and Systems (RSS 2020)*. RSS, 2020.

[21] A. S. Sathya, J. Gillis, G. Pipeleers, and J. Swevers, "Real-time robot arm motion planning and control with nonlinear model predictive control using augmented lagrangian on a first-order solver," in *2020 European Control Conference (ECC)*. IEEE, 2020, pp. 507–512.

[22] M. Vallati and I. Serina, "A general approach for configuring PDDL problem models," in *the 28th International Conference on Automated Planning and Scheduling*, ser. ICAPS'18. AAAI Press, 2018, pp. 431–435.