ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-1537

Partial Order Temporal Plan Merging for Mobile Robot Tasks

Lenka Mudrova[1] and **Bruno Lacerda**[1] and **Nick Hawes**[1]

Abstract. For many mobile service robot applications, planning problems are based on deciding *how* and *when* to navigate to certain locations and execute certain tasks. Typically, many of these tasks are independent from one another, and the main objective is to obtain plans that efficiently take into account where these tasks can be executed and when execution is allowed. In this paper, we present an approach, based on merging of partial order plans with durative actions, that can quickly and effectively generate a plan for a set of independent goals. This plan exploits some of the synergies of the plans for each single task, such as common locations where certain actions should be executed. We evaluate our approach in benchmarking domains, comparing it with state-of-the-art planners and showing how it provides a good trade-off between the approach of sequencing the plans for each task (which is fast but produces poor results), and the approach of planning for a conjunction of all the goals (which is slow but produces good results).

1 INTRODUCTION

Consider a mobile service robot operating in an office building for a long period of time, where it autonomously performs tasks to assist the occupants in their everyday activities. One can imagine a wide array of tasks for such a robot to execute, for example:

- *"Bring me a cup of coffee."*
- *"Check if there are people in office 123."*
- *"Check if the emergency exits are clear."*

Note that these tasks have common properties that one can take advantage of:

1. They require the robot to navigate to certain locations to execute certain actions, i.e., they include *spatial constraints*;
2. The actions associated with them can be executed concurrently. For example, a sensing action is often fixed to a location, but processing and reasoning about the sensed data can be done in parallel with the robot's movement;
3. Their goals are independent, in the sense that executing a certain task ω_i is not a precondition to successfully execute task ω_j. This means that they can be straightforwardly split into as separate planning problems.

Furthermore, even though the goals are independent, the existence of spatial constraints means that there might be "synergies" between the independent plans, i.e. the locations visited while executing task ω_i might also be of use for executing task ω_j.

[1] School of Computer Science, University of Birmingham, UK, email: {lxm210, b.lacerda, n.a.hawes}@cs.bham.ac.uk

In this paper, we build on these insights to present an algorithm that, given a plan for each task, efficiently *merges* them into a single plan for all tasks, interleaving actions which work towards different goals.

Our merging algorithm is based on partial-order planning (POP), a least-commitment search in the space of (partial order) plans. POP presents clear benefits for our robot-oriented merging approach, including:

1. The least-commitment approach of POP yields more "merging points" between plans, when compared to a totally ordered plan;
2. POP presents a flexible approach to temporal planning with durative concurrent actions, allowing parallel action execution;
3. POP produces plans with more flexibility in execution as commitments can be determined at execution time when temporal information is more certain.

The main contributions of this paper are (i) the definition of a class of planning problems that are well-suited for the specification of *execution routines* for mobile service robots; and (ii) a partial order plan merging algorithm that is able to generate a plan for a large number of tasks, while taking advantage of possible synergies between such tasks, thus improving the overall robot's behaviour. For the class of problems we tackle in this paper, our approach is competitive with the performance of state-of-the-art forward chaining planners on benchmarking domains. Furthermore, the use of POP allows us to easily tackle concurrent actions, which allows us to outperform the state-of-the-art forward chaining planners in domains where reasoning about concurrent actions is required.

The structure of the paper is as follows. In Section 2, we provide an overview of state-of-the-art planners, and their limitations for our domain. In Sections 3 and 4 we introduce the background on partial order planning we rely on, formalise the problem we tackle, and describe possible solution approaches. Finally, Sections 5 and 6 present our novel plan merging algorithm, and its evaluation.

2 RELATED WORK

2.1 Temporal planners

In order to compare our proposed algorithm with state-of-the-art temporal planners, we focus on those that successfully participated in the temporal track of the latest International Planning Competition (IPC) in 2014[1]. The 6 following planners participated in the competition: YAHSP3 [30] pre-computes relaxed plans for estimated future states which are then exploited to speed-up the forward state-space search. YAHSP3-MT [31] is a multi-threaded extension of

[1] https://helios.hud.ac.uk/scommv/IPC-14/index.html

YAHSP. The YASHP3 planner is also exploited by another contestant, DAE$_{YAHSP}$ [4]. DAE uses a *Divide-and-Evolve* strategy in order to split the initial problem into a set of sub-problems, and to evolve the split based on the solutions found by its wrapped planner. In general, DAE$_X$ can be used with any planner, with the version we evaluate wrapping YAHSP. Two other participants extend well-known approaches to the temporal domain. First, the temporal fast downward (TFD) planner [10] expands the fast downward algorithm [13]. The search state is extended with a discrete timestamp component, and state expansion can be performed either by inserting an action or by incrementing the timestamp. Second, ITSAT [21] expands a satisfiability checking (SAT) approach to the temporal domain. ITSAT is the only planner from the aforementioned to handle concurrent actions properly.

2.2 Temporal Partial Order Planners

Another important class of temporal planners are those that provide a temporal partial order plan as a solution. Versatile Heuristic Partial Order Planner (VHPOP) [24] is one of the pioneers in this field. It builds on the classical backward plan-space search used by partial order planners, adding to it a set of different heuristics that allow for a more informed choice of which *flaw* to solve, or which plan to explore. The use of these heuristics yields large improvements in terms of speed, when comparing to previous partial order planners. In contrast, the more recent OPTIC [2] planer combines advantages of partial order planning with forward chaining techniques, which are very popular in current planners, due to their speed and scalability.

2.3 Planning & Execution for Service Robots

The CoBot service robots [29] operate in an office building performing several predefined tasks. A server-based architecture [9] manages incoming tasks from a web-based user interface, schedules tasks across several robots [8], and keeps track of task execution. A similar centralised system architecture is used by the mobile service robot Tangy [16] which performs a sequence of predefined tasks, taking the schedules of users into account. This problem is modelled by mixed-integer programming and constraint programming techniques [5]. Mixed-integer programming is also used for scheduling in the integrated control framework presented in [18]. In this work, a stochastic high-level navigation planner provides expectations on travel times between different locations to the scheduling algorithm. In contrast to the previous architectures, robots Rin and Rout use a constraint network [22]. This network is continuously modified by an executor, a monitor and a planner in order to create configuration plans which specify causal, temporal, resource and information dependencies between individual actions. All the above works are based on scheduling approaches, which rely on a coarse-grained model of the environment, where tasks are seen as black-boxes, being pre-specified instead of planned for, and with the scheduler trying to order them such that a set of timing constraints is satisfied. This does not allow for direct reasoning over the possible synergies between different tasks, and the possibility to interleave actions from different tasks in order to minimise execution time.

In recent years, there has also been work aiming at closing the loop between task planning and real world execution on a robot platform. The ROSPlan framework [6] is a general framework that allows for a task planner to be embedded into the Robot Operating System (ROS), a middleware widely used in the robotics community. As a proof of concept, ROSPlan has integrated the POPF planner [7], an ancestor of OPTIC. This integration with execution is also part of our future work, and we plan to explore how our techniques can be integrated in such an execution framework.

Additionally, some modelling languages have been developed with the goal of having a closer integration between planning and execution. Of particular interest are the NDDL [3] and the ANML [25] modelling languages. These are based on the notion of *timelines*, i.e., sequences of *tokens*. A token is a timed predicate that holds within a start and end time. The timeline representation was developed by NASA and used in open-source project EUROPA [1] in order to model and plan for real world problems and to allow a close integration with the execution of such plans. This representation was also exploited in T-REX [17], a model-based architecture for robot control which used a tight integration of planning and execution. Another system closely integrating planning and execution is FAPE [20], built on the ANML language. Unfortunately, these system based on timelines do not have the scalability of the state-of-the-art planning approaches presented above.

2.4 Merging Algorithms

The first planning system to use merging was probably NOAH [23], as described in [11]. In NOAH, three criteria were introduced to handle possible interactions between plans: eliminate redundant preconditions, use existing operators, and optimise disjuncts. NONLIN [26] is also able to recognise if any goal is achievable by an operator already in a plan. If such operator is detected, then ordering constraints and variable bindings are used to change the plan such that the found operator is used to fulfil the goal.

Temporal and conditional plan merging is done in [27] which extends the work of Yang [33]. For two input plans, the algorithm provides a new order of actions while detecting and removing redundant actions, by checking if their effects are already fulfilled by some preceding action.

Related techniques to plan merging are *plan repair* and *plan refinement*. Refinement planning focuses on how to introduce a new action to an existing plan, and was introduced in [15]. Work in [14] uses a *partial plan* to save current refinements. The opposite case, i.e., removing an action from the plan, is handled in *unrefinement planning* [28]. This addresses the *plan repair* problem of changing a plan when it cannot be executed. Despite the fact that it was proved that modifying an existing plan is no more efficient than a full (re)planning in the general case [19], plan repair might still be efficient in certain domains. An example of recent plan refinement is planning for highly uncertain domains, such as underwater robotics [12]. In this case, one plan achieving a subset of tasks is produced. While it is executed, the current state is observed in order to limit uncertainties. If the robot has unexpected available resources, allowing it to perform more tasks, a pre-computed plan achieving another task is inserted into the global plan. Our proposed algorithm combines ideas from aforementioned merging approaches in order to allow flexible execution on a mobile robot.

3 PARTIAL ORDER PLANNING

We start by introducing the definitions for POP that will be used throughout the paper. For a thorough overview of POP see [32]. Our merging algorithm assumes that plans have already been generated, so all actions we deal with are grounded. Thus, we omit details about lifted actions and bindings when describing the planning problems.

We start by defining a task in our framework. A task domain is a set $D = \{f_1, .., f_n\}$ of atomic formulas (atoms); literals – formulas and their negations $L_D = \{f_1, \neg f_1, \dots\}$ – are used to describe a given state of the world. A state in this domain is represented by $L \subseteq L_D$, such that either f or $\neg f$ are in L. A literal l is satisfied in L if $l \in L$.

A durative action $a \in A$ consists of its start point $a_\vdash$ and end point $a_\dashv$, and we define the sets $A_\vdash$ and $A_\dashv$ of action start points and end points, respectively. Preconditions $pre(a)$ of a durative action a are a set of *timed* literals, which are literals which must hold at a specific action point. We recognise the *at start* literal which must hold at the action start point $a_\vdash$, the *at end* literal which holds at the action end point $a_\dashv$ and the *over all* literal which must hold during whole action. Hence, if we refer to $pre(a_\vdash)$ we mean only those literals which are meant to hold *at start* or *over all*, etc. A set of action effects $\mathit{eff}(a)$ is a set of timed literals as well but only *at start* or *at end* extensions are assumed, as an *over all* effect is same as *at start* effect. The duration of an action is $d(a) \in \mathbb{R}$.

Finally, a planning problem is defined as $P = (I, G)$, where $I \subseteq L_D$ is the initial state, and $G \subseteq L_D$ is the goal. We say that a state L achieves the goal if $G \subseteq L$. A task is then defined as $\omega = \langle D, A, P\rangle$.

A partial order plan (POP) is a tuple $\pi = \langle \mathcal{A}, \mathcal{L}, \mathcal{O}\rangle$, where:

- $\mathcal{A}$ is a set of actions;
- $\mathcal{L}$ is a set of causal links. A causal link $a_j \xrightarrow{l} a_k$ represents that literal $l \in pre(a_k)$ is fulfilled as an effect of action a_j;
- $\mathcal{O}$ is a set of ordering constraints defining a partial order on the set $\mathcal{A}$. An ordering constraint $a_j \prec a_k$ represents that action a_j must finish before action a_k can start.

Given a causal link $a_j \xrightarrow{l} a_k$, we refer to a_j as the *producer* of literal l and to a_k as the *consumer* of literal l. Given a POP π, an open condition $\xrightarrow{l} a_j$ means that literal $l \in pre(a_j)$ has not yet been linked to the effect of any action. An unsafe link (or a threat) is a causal link $a_j \xrightarrow{l} a_k$ such that there is an action $a_m \in \mathcal{A}$ that could possibly be ordered between a_j and a_k and threatens $a_j \xrightarrow{l} a_k$ by having $\neg l \in \mathit{eff}(a_m)$. The set of flaws of a POP π is given by the union of its open conditions and unsafe links. A POP planner searches through the space of POPs trying to resolve all flaws of π. To do that, the planner tries to close open conditions by adding new actions to $\mathcal{A}$ and new causal links to $\mathcal{L}$, and solve threats by adding new orderings to $\mathcal{O}$ to make sure that the threatening action a_m does not occur between the unsafe link $a_j \xrightarrow{l} a_k$. This can be done by, for example, *promotion*, i.e., adding the constraint $a_k \prec a_m$ to $\mathcal{O}$, or *demotion*, i.e., adding $a_m \prec a_j$ to $\mathcal{O}$. In this work, we use the VHPOP planner as described in Section 2.2.

4 PROBLEM DEFINITION AND SOLUTION APPROACHES

In this paper, we are interested in solving the problem of finding a POP for a given set of input tasks. More specifically, given a set of tasks $\Omega = \{\omega_1, \dots, \omega_n\}$, where $\omega_i = \langle D_i, A_i, (I_i, G_i)\rangle$ and the initial states of each problem are mutually consistent (I_j is consistent with I_k if for all $l \in L_{D_j} \cap L_{D_k}$, $l \in I_j$ if and only if $l \in I_k$), we want to find a plan that achieves $G_1, ..., G_n$.

There are several ways of solving such a problem. In this section, we present three different approaches: *unifying*, *sequencing* and *plan merging*. We argue that the merging approach provides a good trade-off between the plan quality of the unifying approach and the efficiency of the sequencing approach, hence, in the next section we present a plan merging algorithm to solve our problem.

4.1 Unifying planning algorithm

This approach relies of *unifying* the set of tasks into a single task, i.e., $\omega = \left\langle \bigcup_{i\in\{1\dots n\}} D_i, \bigcup_{i\in\{1\dots n\}} A_i, (\bigcup_{i\in\{1\dots n\}} I_i, \bigcup_{i\in\{1\dots n\}} G_i)\right\rangle$ Then, one can use an appropriate planning algorithm to find a solution for the single unified task. While this approach can more easily take advantage of relations between goals in different tasks (e.g., two tasks that should be executed in the same location), it can suffer from scalability issues as finding a plan for the unified task can be much harder than finding plans for each individual task by itself. This approach is used by most planners, such as VHPOP, OPTIC and all presented planners from IPC 2014. Generally, the unified tasks are modelled a priori and passed directly as an input to such systems.

4.2 Sequencing planning algorithm

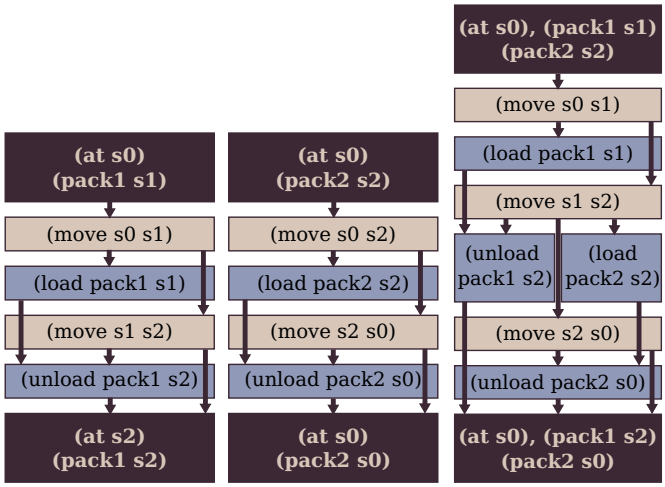
This approach generates a set of independent plans $\Pi_\omega = \{\pi_1, \dots, \pi_n\}$, one for each task ω_i, and then *sequences* them to create a single final plan. For the resulting plan to be valid, one needs to decide on an ordering of the tasks and then modify the initial states of each task according to the final state of the plan for the preceding task. The ordering of the tasks can be done using a *scheduling algorithm* that can take into account extra timing constraints on the execution of tasks. This approach is common for mobile service robots, e.g., [29, 18], due to its simplicity and efficiency. This is because the planning problems to be solved when planning for the tasks independently will in general be much smaller than the single unified one problem. However, simple sequencing comes at the price of plan quality: this approach does not allow for the interleaving of actions from plans for different tasks, taking advantage of synergies between them.

4.3 Merging planning algorithm

This approach combines both aforementioned methods. It also plans for tasks separately, obtaining $\Pi_\omega = \{\pi_1, \dots, \pi_n\}$ but then it reasons over each plan, *merging* them together into a better plan than the one obtained by simple sequencing. The final plan π_f consists of parts of the task plans in Π_ω and newly created plans Π_{join} which are used in order to connect these parts such that the final plan is free of flaws. Furthermore, while the merging procedure adds an overhead at plan generation time when compared to the sequencing approach, it allows us to find synergies between plans for different tasks, interleaving execution for different goals. A typical benefit of this approach in the mobile robot domain is the possibility to execute actions from different tasks when these actions share a common location. The algorithm we present in the next section follows this approach.

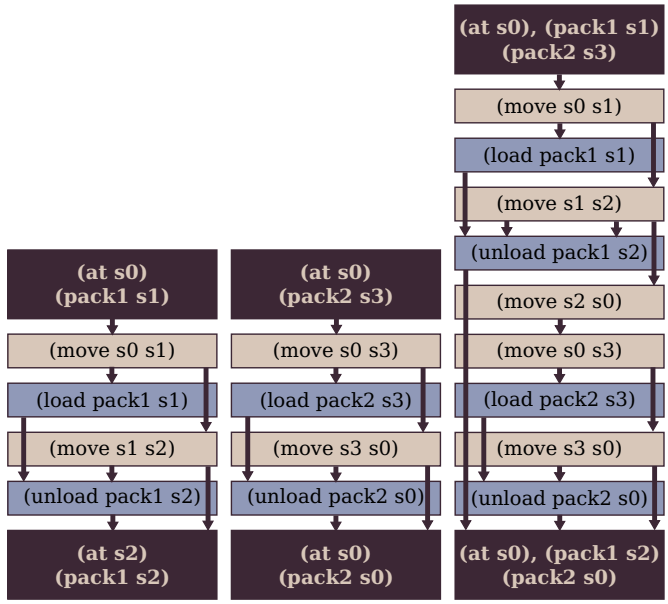
5 PROPOSED ALGORITHM

In this section, we present our merging algorithm. Before we describe it, we need to address an issue that can hinder the performance of the merging algorithm, and present a solution for it.



(a) Plan π_{1a} (b) Plan π_{2a} (c) Plan π_{fa}

Figure 1. Single plans π_{1a}, π_{2a} for tasks ω_1, ω_2 respectively and the merged plan π_{fa}. The arrows represent the causal links between actions. The order of actions in the plan corresponds to the orderings, for example two concurrent actions are parallel in the final plan.



(a) Plan π_{1b} (b) Plan π_{2b} (c) Plan π_{fb}

Figure 2. Single plans π_{1b}, π_{2b} for tasks ω_1, ω_2 respectively and the merged plan π_{fb}. The arrows represent the causal links between actions. The order of actions in the plan corresponds to the orderings.

5.1 Dependency Caused by External Constraints

5.1.1 Problem Illustration

As stated in the introduction, we assume independent tasks, i.e., tasks where the goals can be partitioned and for which the execution of a plan for task ω_i is not a precondition to successfully execute task ω_j. However, these tasks can become dependent due to external constraints. In mobile robots domains, these are typically spatial constraints. We will illustrate this problem on the following example.

DeliveryBot A mobile robot delivers packages around a building. A robot can *move between locations* (with duration 10), *load packages* (with duration 2), and *unload packages* (with duration 2). The robot receives two tasks:

1. ω_1: *"Deliver package1 to location s2."*
2. ω_2: *"Deliver package2 to location s0."*

Assume the initial state is $I = \{$*(at s0), (pack1 s1), (pack2 s2)*$\}$, and a partial order planner produces optimal plans π_{1a} and π_{2a}, as depicted in Fig. 1a and Fig. 1b. An example of a final merged plan, π_{fa}, with makespan 38 is also given in Fig. 1c. Notice that action *(move s0 s2)* from plan π_{2a} is not used as its effects are satisfied by action *(move s1 s2)* from plan π_{1a}. However, if the initial state would be $I = \{$*(at s0), (pack1 s1), (pack2 s3)*$\}$, a partial order planner would produce the plan π_{2b} which is again optimal, see Fig. 2b. In this case, action *move(s0 s3)* will need to be merged as well as its effects are not satisfied. Hence, the output plan has makespan 58, see Fig. 2c. However, an optimal plan will not contain *(move s2 s0), (move s0 s3)* and will instead directly use action *(move s2 s3)*. Thus, the optimal plan for the two goals has makespan 48.

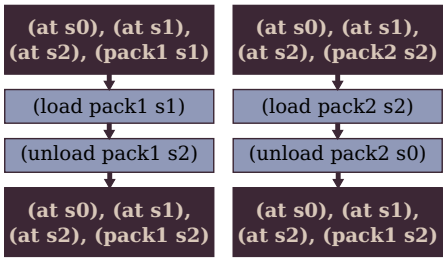
The problem during merging is that an action in an input plan is linked to a preceding state which contains all its preconditions. However, during merging, the action might be chosen to be merged into the final plan in a state which does not achieve all of the action's preconditions. In such cases, *filling actions* (i.e., actions whose effects change the current state to contain all needed preconditions) need to be added to the final plan before the candidate action is merged. This extends the makespan of the final plan, hence we would like to minimise the occurrence of such filling actions. Furthermore, we note that actions related to the external constraints cause unnecessary joining actions. For example, action *(move s2 s0)* in plan π_{fb} is a filling action, required for action *(move s0 s3)* from the original plan to be merged.

As discussed in the introduction, actions related to robot movement (in this case, action *move*) are typically a significant contributor to the makespan of plans generated on a mobile service robot domain. Hence, minimising their occurrence in the plan generally also allows for reducing on the makespan. Therefore, addressing these dependencies caused by external constraints before merging can lead to significant improvements to the quality of the merged plans.

5.1.2 Preprocessing External Constraints

In order to address the problem described above, we create relaxed planning problems that have their initial states extended by literals which satisfy the external constraints. Therefore the resulting plans for these relaxed problems will not contain any external constraints. By removing the external constraints from the input problems, we allow more freedom for the merging algorithm o merge them together, resulting in a final plan with better makespan.

In mobile service robot domains, the external constraints are related to the position of the robot. Therefore in our in the DeliveryBot example we add *(at s0), (at s1), (at s2)* to the initial states for the individual task plans, see Fig. 3a and Fig. 3b. We will address the automated detection of external constraints in future work.



(a) $\pi_1^\times$ (b) $\pi_2^\times$

Figure 3. Relaxed plans created by adding all locations to the initial states.

5.2 Plan Merging Algorithm

Given a set of relaxed plans, the proposed algorithm for plan merging $POMer_X$ incrementally extracts actions from the input plans and greedily adds an action start point $a_\vdash$ or an action end point $a_\dashv$ per iteration to the merged plan. Action end points are added depending on their temporal constraints, but action start points are added based on whether they require a *joining* subplan. A joining subplan is required by an action start if its preconditions are not satisfied by the effects of actions already merged into the final plan. This is caused by two factors. First, the relaxation of the input plans removes external constraints that need to be addressed during plan merging. Second, the greedy merging of actions might cause for an interleaving of actions from different plans that yield certain action inapplicable. We refer to an action requiring a *joining* plan as *inapplicable* in the current state.

If there exists action starts which do not require a *joining* plan, the algorithm picks one of them to be merged. Otherwise, $POMer_X$ calls its wrapped POP planner X in order to find the joining subplan that satisfies the preconditions and connects the current state of the merged plan to the preconditions of the new action start point. Then, the greedy decision is made over all *joining* subplans, with a single joining subplan with the minimal duration being chosen; its actions are then extracted and picked to be merged the same way as actions from the input plans.

Furthermore, $POMer_X$ allows for backtracking when the greedy choices result in an intermediary plan that does not satisfy the temporal constraints or causal links of the input plans. Thus, while not optimal, $POMer_X$ is guaranteed to find a solution for the merging, if one exists.

We now describe the main steps of the algorithm. Let $\Omega = \{\omega_1, ..., \omega_n\}$ be a set of tasks, with $\omega_i = \langle D_i, A_i, (I_i, G_i)\rangle$, and $\Pi^\times = \{\pi_1^\times, ..., \pi_n^\times\}$ be a set of relaxed plans, where $\pi_i^\times$ is the plan obtained for some relaxation $\langle D_i, A_i, (I_i^\times, G_i)\rangle$ of ω_i. The input of Algorithm 1 is the initial state of the (unrelaxed) global problem $\mathcal{I} = \cup_{i\in\{1,...,n\}} I_i$, the set $\Pi^\times$ of relaxed plans, and the set $\mathcal{G} = \cup_{i\in\{1,...,n\}} G_i$, of goal states for each task. The algorithm then calculates a merged plan π_s such that all goals $G_1, ..., G_n$ are satisfied, when applying plan π_s to the initial state $\mathcal{I}$. To construct the final plan π_s, the algorithm searches over *merging states* of the form

$$S = \langle t, L, \mathcal{A}^+, \mathcal{A}^-, Q_\vdash, Q_\dashv, L_{block}, \Pi_{join}, \Pi_\omega^\times, \pi_s\rangle \qquad (1)$$

where:

- t is duration of the current plan π_s;
- $L = \{F^+, \neg F^-\}$ is the set of literals that hold in S, where sets F^+, F^- contains atoms that hold and do not hold in the current state, respectively.
- A^+ is the set of *achievers* of all atoms that hold in the current state. The achiever of f, denoted $A^+(f)$, is the last action a added to π_s such that $f \in \mathit{eff}(a)$.
- A^- is the set of *deleters* of all atoms that do not hold in the current state. The deleter of f, denoted $A^-(f)$, is the last action a added to π_s such that $\neg f \in \mathit{eff}(a)$.
- $Q_\vdash \subseteq A_\vdash$ is a queue of action start points that are extracted from the input plans to be merged into plan π_s;
- $Q_\dashv \subseteq A_\dashv \times \mathbb{R}$ is a queue of action end points for actions for which the start point has already been merged into π_s. Each end point is of the form $(a_\dashv, t_{a_\vdash} + d(a))$, where $t_{a_\vdash}$ is the time point when the start of a was merged into π_s;
- $L_{block} \subseteq L \times A_\vdash$ is a set of *blocked* literals. Each blocked literal $(l_{block}, a_\vdash) \in L_{block}$ is such that the starting point $a_\vdash$ of action a has been already merged to π_s in some previous state, the end point $a_\dashv$ of a is not yet merged, $l_{block} \in pre(a)$, and it must hold *over all* duration of the action. Therefore, l_{block} must hold until $a_\dashv$ is merged, thus removing the blocked literal; the validity of blocked literals cannot be changed by other actions in the merging process.
- Π_{join} is a set of *joining* POPs satisfying preconditions of actions in $Q_\vdash$.
- $\Pi_\omega^\times$ is the set of all input plans for all tasks.
- $\pi_s = \langle \mathcal{A}_{\pi_s}, \mathcal{L}_{\pi_s}, \mathcal{O}_{\pi_s}\rangle$ is the POP that reaches the current state.

At each step on its main loop, Algorithm 1 starts by analysing the queue of action end points $Q_\dashv$ to check if there are action end points that must be merged into the plan at the current state, given their temporal constraints. If so, the action end point is merged into π_s. Otherwise, the algorithm proceeds by adding to $Q_\vdash$ the starts points of actions a in each relaxed input plan for which the following three conditions hold. First, a has not been merged before; second, the *producing* actions in the input causal links where a is a *consumer* have already been merged; and third, the actions that must be ordered before a are already merged (lines 8–14). The merged actions, links and orderings $\langle \mathcal{A}_m, \mathcal{L}_m, \mathcal{O}_m\rangle$ are obtained by a method *merged-subplan*($\pi_i^\times$) which for a given plan returns a subplan that is already *merged* in the current state S, i.e., $\mathcal{A}_m \subseteq \mathcal{A}_{\pi_s}, \mathcal{L}_m \subseteq \mathcal{L}_{\pi_s}, \mathcal{O}_m \subseteq \mathcal{O}_{\pi_s}$. If all merged subplans are equal to the input plans, the algorithm successfully merged all the plans and it returns the final plan. (line 17). Otherwise, if $Q_\vdash$ is not empty, method *pickActionStart*($Q_\vdash$) (see Algorithm 2) will choose an action start point that is not affecting any of the blocked literals in L_{block}. All action starts affecting blocked literals are removed from $Q_\vdash$.

In the cases where $Q_\vdash$ is empty, i.e., no action start points can be extracted from the input plans, or all action start points were removed due to affecting blocked literals, there is no action start point that can be merged at the current state. Hence, the next action end point to be merged (i.e., the one with the earliest deadline) is merged instead, if one exists. If no action end point exists either, then we have reached a state for which it is not possible to merge actions while satisfying the input plans constraints. Therefore, the algorithm backtracks to the parent state, removing the last merged action from the plan and propagates it as an invalid choice. If the initial state is reached by backtracking, it means that the plans cannot be merged while maintaining their individual constraints, and the algorithm outputs that there is no solution.

Algorithm 2 chooses one action start point $a_\vdash \in Q_\vdash$ to be merged into the final plan. It starts by checking which actions in $Q_\vdash$ are *in-*

Algorithm 1 POMer$_X(\Pi^\times, \mathcal{I}, \mathcal{G})$

1: $S = \langle 0, \mathcal{I}, \mathcal{A}_0^+, \mathcal{A}_0^-, \emptyset, \emptyset, \emptyset, \emptyset, \check{\Pi}_\omega^\times, \langle \{start\}, \emptyset, \emptyset \rangle \rangle$
2: **while true do**
3: **if** $\exists (a_\dashv, t_\dashv) \in Q_\dashv$ such that $t = t_\dashv$ **then**
4: remove $(a_\dashv, t_\dashv)$ from $Q_\dashv$
5: merge($a_\dashv$)
6: **else**
7: $allMerged$ = **true**
8: **for** $\pi_i^\times = \langle \mathcal{A}^\times, \mathcal{L}^\times, \mathcal{O}^\times \rangle \in \Pi^\times$ **do**
9: $\langle \mathcal{A}_m, \mathcal{L}_m, \mathcal{O}_m \rangle$ =merged-subplan($\pi_i^\times$)
10: **if** $\langle \mathcal{A}_m, \mathcal{L}_m, \mathcal{O}_m \rangle \neq \langle \mathcal{A}^\times, \mathcal{L}^\times, \mathcal{O}^\times \rangle$ **then**
11: $allMerged$ = **false**
12: **end if**
13: $\{a \in \mathcal{A}^\times \setminus \mathcal{A}_m \mid \forall (a' \xrightarrow{l} a \in \mathcal{L}^\times), \mid \forall (a' \prec a \in \mathcal{O}^\times),\ a' \in \mathcal{A}_m^\times\}$
14: $Q_\vdash = Q_\vdash \cup a_\vdash$
15: **end for**
16: **if** $allMerged$= **true then**
17: **return** π_s
18: **end if**
19: **if** $Q_\vdash \neq \emptyset$ **then**
20: $\{a, Q_\vdash\}$=pickActionStart($Q_\vdash$)
21: **if** $a \neq \emptyset$ **then**
22: $merge(a)$
23: **continue**
24: **end if**
25: **end if**
26: **if** $Q_\vdash = \emptyset$ **then**
27: **if** $Q_\dashv \neq \emptyset$ **then**
28: pick $a_\dashv \in Q_\dashv$ with the soonest deadline
29: merge($a_\dashv$)
30: **else**
31: S = backtrack()
32: **end if**
33: **else**
34: go to line 19
35: **end if**
36: **end if**
37: **end while**

applicable. An action start $a_\vdash$ is *inapplicable* in the current state if at least one of its preconditions do not hold in the current state. This can happen if another action deleting literal l was merged after the achiever of l; or the literal was relaxed in the input plan. Thus, in line 1, we build the set of *applicable* of action starts, i.e., the set of action start points that are not *inapplicable* at the current search state. If there are no applicable actions at the current state, we generate, for each $a_{vdash} \in Q_\vdash$, *joining* plans. To do so, we use a call to the wrapped POP planner, defining the current set of literal L as the initial state, and the preconditions of the action as the goal state (line 3). From the set of joining plans, we then choose the one that has the shortest duration, and insert its applicable actions to the *applicable* set (lines 4 and 5).

Then, we remove from *applicable* and $Q_\vdash$ all actions that affect the *blocked* literals in L_{block}. Note that the actions removed from $Q_\vdash$ will, in some future state, be again extracted from the input plans and will be potentially merged into the final plan. If after removing the actions that affect L_{block}, *applicable* is empty, then the algorithm returns an empty action, else it non-deterministically picks and returns an action start point $a_\vdash \in applicable$.

Algorithm 2 pickActionStart($Q_\vdash$)

1: $applicable = \{a_\vdash \in Q_\vdash \mid pre(a_\vdash) \subseteq L\}$
2: **if** $applicable = \emptyset$ **then**
3: $\Pi_{join} = \{X(D, A, \langle L, pre(a_\vdash) \rangle) \mid a_\vdash \in Q_\vdash\}$
4: pick $\pi_{join} = \langle \mathcal{A}_{join}, \mathcal{L}_{join}, \mathcal{O}_{join} \rangle$ from Π_{join} that has the shortest duration
5: $applicable = \{a_\vdash \in \mathcal{A}_{\vdash join} \mid pre(a_\vdash) \subseteq L\}$
6: **end if**
7: remove from $Q_\vdash$ and $applicable$ all actions a that do not affect L_{block}, i.e., $\forall l \in \mathit{eff}(a) : (\neg l, .) \notin L_{block}$
8: **if** $applicable = \emptyset$ **then**
9: **return** $\{\emptyset, Q_\vdash\}$
10: **else**
11: pick action a from $applicable$
12: **return** $\{a, Q_\vdash\}$
13: **end if**

Finally, Algorithm 3 updates the current state S depending on the chosen action point, i.e., an action start or end, and merges it into the plan π_s. It proceeds as follows. First, if an action end is being merged, the time is updated by the corresponding time deadline of the action. An action start cannot move the time forward. Then, the set of active literals, achievers and deleters are updated by the effects of the action point; additionally the set of blocked literals is updated by preconditions (line 11 – 16). The action point is removed from the particular queue and added to the set of actions A_{π_s}. For each precondition of the action point, new causal link and corresponding ordering is added between achiever or deleter (for negative literals) and the merged action point. Finally, the added action point might be ordered in parallel to another action and it might threaten an existing causal link (line 22). In such a case, the action point is *promoted*.

Algorithm 3 merge(a_{inst})

1: $t = (a_{inst} \in Q_\dashv)\ ?\ t = Q_\dashv[a_{inst}]$
2: $L = L \cup \mathit{eff}(a_{inst})$
3: **for all** $f \in \mathit{eff}^+(a_{inst})$ **do**
4: $A^+(f) = a_{inst}$
5: **end for**
6: **for all** $f \in \mathit{eff}^-(a_{inst})$ **do**
7: $A^-(f) = a_{inst}$
8: **end for**
9: $(a_{inst} \in Q_\vdash)\ ?\ Q_\vdash = Q_\vdash \setminus a_{inst}$
10: $(a_{inst} \in Q_\dashv)\ ?\ Q_\dashv = Q_\dashv \setminus a_{inst}$
11: **for all** $l \in pre(a_{inst})$ **do**
12: **if** $a_{inst} \in Q_\vdash$ **and** l is *over all* **then**
13: $L_{block} = L_{block} \cup l$
14: **end if**
15: **if** $a_{inst} \in Q_\dashv$ **and** l is *over all* **then**
16: $L_{block} = L_{block} \setminus l$
17: **end if**
18: **end for**
19: $\mathcal{A}_{\pi_s} = \mathcal{A}_{\pi_s} \cup a_{inst}$
20: $\forall l \in pre^+(a_{achi}) : \mathcal{L}_{\pi_s} \cup \langle A^+(l) \xrightarrow{l} a_{inst} \rangle$, $\mathcal{O} \cup \langle A^+(l) \prec a_{inst} \rangle$
21: $\forall l \in pre^-(a_{inst}) : \mathcal{L}_{\pi_s} \cup \langle A^-(l) \xrightarrow{l} a_{inst} \rangle$, $\mathcal{O} \cup \langle A^-(l) \prec a_{inst} \rangle$
22: A_{threat} = threats(a_{inst})
23: $\forall a_{threat} \in A_{threat} : \mathcal{O}_{\pi_s} \cup \langle a_{threat} \prec a_{inst} \rangle$

5.3 Example

The flow of the algorithm is illustrated on the DeliveryBot example. The input to the algorithm is the conjunction of the original initial states, i.e., $\mathcal{I} = \{$*(at s0)*$\}$, the conjunction of the goals $\mathcal{G} = \{$*(pack1 s2), (pack2 s0)*$\}$ and the extracted plans $\pi_1^\times, \pi_2^\times$, see Fig. 3a and Fig. 3b. At the first iteration, the *merged-subplans* are empty, hence Algorithm 1 on line 13 chooses from all actions in the input plans. First, actions from the input plan on Fig. 3a are being extracted; action *(load pack1 s1)* has two preconditions: *(at s1), (pack1 s1)*. The achiever of both preconditions is the *start* action in the relaxed plan. Hence, this action is extracted in order to be merged. Then, *(unload pack1 s2)* action is tested. This action has two preconditions as well: *(pack1 truck)* and *(at s2)*. The achiever of the first precondition is the action *(load pack1 s1)*, however that action is not yet merged in plan π_s. Hence, action *(unload pack1 s2)* cannot be extracted to be merged. The same reasoning is done for the second plan $\pi_2^\times$.

After that, Algorithm 2 proceeds by extracting *applicable* actions from $Q_\vdash = \{$*(load pack1 s1), (load pack2 s2)*$\}$. Even though both actions have satisfied preconditions in the input relaxed plan, they are not satisfied in the current set of literals $L =$*(at s0)* due to relaxation. Therefore, the *applicable* set is empty and the wrapped planner X is called in order to obtain *joining* plans to achieve the preconditions of actions in $Q_\vdash$. The plan $\pi_{join-1} = \langle\{a_{1-1} =$*(move s0 s1)*$\}, \{start \xrightarrow{(at\ s0)} a_{1-1}\}, \{start \prec a_{1-1}\}\rangle$ is obtained, as is the similar π_{join-2} containing only action *(move s0 s2)*. In this particular case, the plans contain only a single action but in general, they can contain more. The temporary plan with the minimal duration is chosen; in this example, both plans have same duration, hence one plan is chosen randomly and *applicable* actions are extracted from the plan, i.e. *applicable* =*(move s0 s1)*. Finally, one action start point, in this case *(move s0 s1)*$\vdash$ is merged into the plan π_s.

Now we illustrate the meaning of *blocked* literals. In the third iteration, the action *(load pack1 s1)* is chosen; it has precondition *(at s1)* which must be valid *over all* duration of the action. Hence, $L_{block} =$ *(at s1)*. In the next iteration, actions' start *(move s1 s2)*$\vdash$ from the *applicable* set has $\neg$*(at s1)* as an effect which will change the blocked literal. Therefore, it cannot be chosen and set $Q_\vdash$ will become empty. Thus, Algorithm 1 proceeds to line 28 and chooses *(load pack1 s1)*$\dashv$ from the queue $Q_\dashv$ and merges it.

6 EVALUATION

We have developed a version of our algorithm POMer$_{VHPOP}$ which embeds the VHPOP planner [24] for plan generation for individual tasks[2]. In this section, we evaluate the POMer$_{VHPOP}$ algorithm and compare it with other temporal planners based on plan quality, measured using the makespan of the found solution, and scalability. For each found plan, we run VAL, the validator of PDDL plans [3] in order to ensure that the plan is valid for domain and problem. The evaluated planners maximum memory usage was limited to 8 GB. All evaluation was run on Lenovo ThinkPad E-540 with Intel i74702MQ Processor (6MB Cache, 800 MHz).

6.1 Domains and problems

We evaluate using domains taken from IPC 2014. However, we generate our own planning problems in these domains, as our algorithm is based on the assumptions that tasks, i.e., goals in problems, are independent. Moreover, we assume only a single agent performing the actions.

6.1.1 Drivelog domain

The Drivelog domain is the IPC 2014 domain closest to our main focus of mobile service robots as it has spatial constraints. In order to generate problems, we take the hardest problem from IPC 2014, i.e., problem P_{23}, and modify it so that it has a single agent, i.e., one truck with the driver already boarded. Then, we split the 23 goals for package placing into 23 single tasks. For the merging algorithm the initial state of these single tasks is extended by adding all atoms related to the spatial constraints, i.e., all *(at ?loc)* where *?loc* stands for any location in the problem.

6.1.2 TMS domain

This domain is another benchmarking domain for IPC 2014 which requires *concurrent* actions, a type of problem for which POP problems are especially suited. Even though this domain is about producing ceramic pieces, we choose it in order to demonstrate the capability of POMer$_{VHPOP}$ to handle concurrent actions. In this domain, a kiln represents the agent. Hence, our problems contain initial state that a kiln is always ready. We take the hardest problem from IPC 2014 and from it create 17 problems. The smallest problem contains 2 goals and the largest 50 goals.

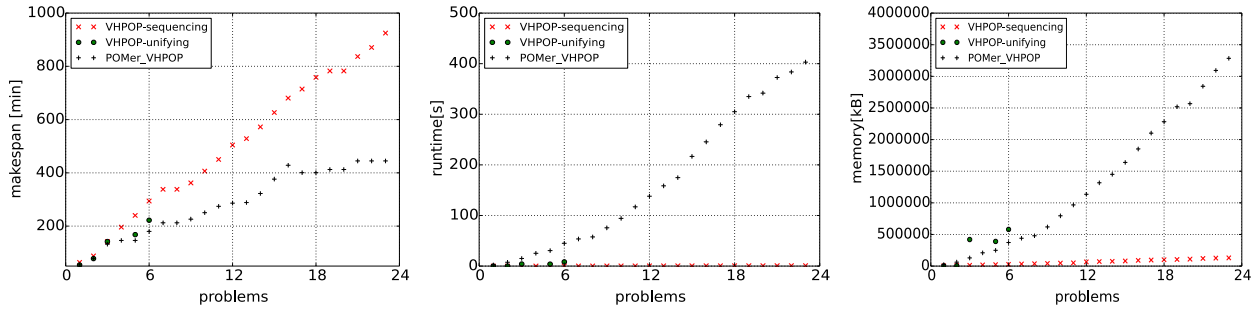
6.2 POMer$_{VHPOP}$ in comparison to VHPOP

First, we analyse how our proposed algorithm improves over its wrapped planner, in this case VHPOP. Thus we compare three algorithms: POMer$_{VHPOP}$, VHPOP used to solve the unified problem, and VHPOP used to solve the sequencing problem. All algorithms were run on problems for Drivelog domain for 30 min and could use 8 GB of memory. The makespans are depicted in Fig. 4a. VHPOP-unifying is able to find a solution for only five problems before it reaches the memory limit. We also report on time and memory consumed, see Fig. 4b and Fig. 4c, respectively. As expected, VHPOP-unifying consumes the most memory for most of the cases and in problem 4, and problems 6-23, it does not find a solution before the limit of 8 GB is reached. In contrast, the sequencing approach is the fastest and the most memory efficient however it always finds the worst makespan. For the largest problem, the makespan found by the sequencing approach is double the one found by POMer$_{VHPOP}$. This means that if the makespan is expressed in duration POMer$_{VHPOP}$ saves about 460 min in the biggest problem comparing to the fast sequencing approach even though it takes up to 7 min to provide a solution. To summarise, we can state that our merging algorithm wrapped around VHPOP significantly improved scalability of standalone VHPOP. As POMer$_{VHPOP}$ is not yet optimised, it is the slowest approach.

6.3 POMer$_{VHPOP}$ in comparison to IPC planners

This evaluation is focused on comparing properties of our proposed algorithm POMer$_{VHPOP}$ with the state of the art planners from IPC 2014, such as DAE$_{YAHSP}$, YAHSP3-MT, TFD and ITSAT, as described in Section 2. Additionally, we also compare to the POP-based OPTIC planner [2] and to VHPOP using the sequencing solution.

[2] Available at https://github.com/mudrole1/POMer
[3] http://www.inf.kcl.ac.uk/research/groups/PLANNING/



(a) Makespan (b) Time (c) Memory

Figure 4. Comparision of POMer$_{VHPOP}$, VHPOP-unifying and VHPOP-sequencing.

6.4 Drivelog domain

For each problem from the Drivelog domain, the aforementioned planners were run for the duration POMer$_{VHPOP}$ required for the same problem, see Fig. 4b. In order to express a quality of the found makespan, we introduce estimates of the best and the worst makespan. As the worst estimate, we use makespan found by VHPOP in sequencing approach and as the best estimate, we run DAE$_{YAHSP}$ for 30 minutes. Fig. 5 shows the recorded makespan for each problem. Even though the makespan is not a continuous function, we visualise the worst and the best estimates as a line in order to highlight these limits.

Note that DAE$_{YAHSP}$ struggles in problems 7 and 11 to provide a good solution. In both cases, the found solution is even slightly worse than the worst estimate. Hence, we exclude these problems from the following analysis. Our algorithm is better than the best estimate in three problems by total difference 46.9. This means that an average difference per plan is 15.63 units in which makespan is recorded. As all packages must be loaded and unloaded in both plans, the POMer$_{VHPOP}$ has only two options how to find better plan - place loading and unloading actions concurrently or find better path between locations. In 20 cases, POMer$_{VHPOP}$ found worse solution than the best estimate by a total difference of 327.88, or 16.39 average difference. Most of these cases were problems with more goals, which is expected as the greedy heuristics are driven to local optima more often in bigger problems.

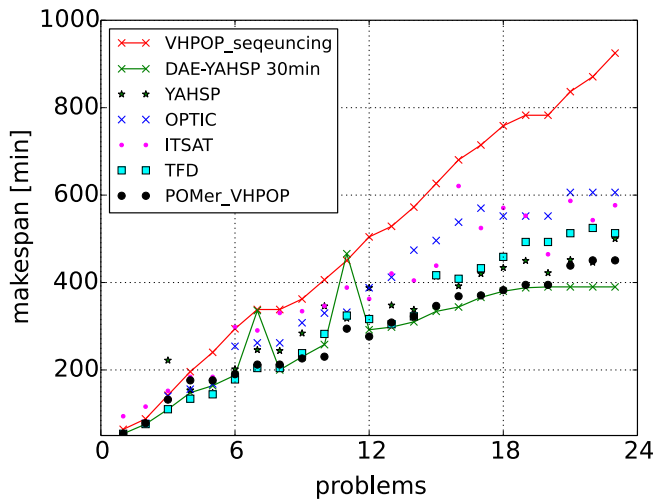


Figure 5. Makespans for problems in Drivelog domain.

6.5 TMS domain

Even though, planners DAE$_{YAHSP}$, YAHSP and TFD perform very well in Drivelog domain, they are unable to find valid solutions in TMS domain as they do not handle concurrent actions correctly. As result, we comparing POMer$_{VHPOP}$ with only OPTIC, ITSAT and the original VHPOP. However an interesting phenomena occurs for our problems: all planners find almost the same makespans. This phenomena occurs due to a fact that a kiln used for baking ceramic has no resource limits. Thus all the ceramic pieces can be baked in parallel.

7 CONCLUSION

We presented an approach for merging of partial order plans especially suited for mobile service robots that need to execute tasks at different locations in an environment. The approach is based on first solving relaxed problems for each individual task, and then perform search over the solutions for these relaxed problems, stitching them together in a way that takes advantage of the synergies between the different tasks. We provided an evaluation of our approach on two benchmarking domains, showing that, for the class of problems we are interested in, it is competitive with state-of-the-art temporal planners. Furthermore, it illustrated our approaches flexibility, as it can perform well in the two domains we analysed, while the other approaches have issues in at least one of the domains.

Future work includes developing an automatic relaxation of the individual problems, and tackling issues related to the execution of the plans we are generating in a mobile robot. This includes closing the loop between plan generation and execution, for which we feel partial order plans are better suited than totally ordered ones, and tackle other common issues for service robotics, such as timing constraints on task execution, the uncertainty inherent to execution in the real world, or merging of plans for new tasks arriving during execution.

REFERENCES

[1] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, E. Remolina P. Morris, J. Ong, T. Smith, and D. Smith, 'Europa: A platform for ai planning, scheduling, constraint programming, and optimization',

in *Proc. of 4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

[2] J. Benton, Amanda Jane Coles, and Andrew Coles, 'Temporal planning with preferences and time-dependent continuous costs', in *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, (2012).

[3] Sara Bernardini and David E. Smith, 'Developing domain-independent search control for europa2', in *ICAPS-07 Workshop on Heuristics for Domain-independent Planning*, (2007).

[4] Jacques Bibaï, Pierre Savéant, Marc Schoenauer, and Vincent Vidal, 'An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning', in *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pp. 18–25, Toronto, ON, Canada, (May 2010). AAAI Press.

[5] K. E. C. Booth, T. T. Tran, G. Nejat, and J. C. Beck, 'Mixed-integer and constraint programming techniques for mobile robot task planning', *IEEE Robotics and Automation Letters*, **1**(1), 500–507, (Jan 2016).

[6] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcís Palomeras, Natàlia Hurtós, and Marc Carreras, 'Rosplan: Planning in the robot operating system', in *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pp. 333–341, (2015).

[7] A. J. Coles, A. I. Coles, M. Fox, and D. Long, 'Forward-chaining partial-order planning', in *The International Conference on Automated Planning and Scheduling (ICAPS-10)*, (May 2010).

[8] Brian Coltin and Manuela M. Veloso, 'Online pickup and delivery planning with transfers for mobile robots', in *Proc. of 2014 IEEE Int. Conf. on Robotics and Automation (ICRA)*, (2014).

[9] Brian Coltin, Manuela M. Veloso, and Rodrigo Ventura, 'Dynamic user task scheduling for mobile robots.', in *Proc. of 2011 AAAI Workshop on Automated Action Planning for Autonomous Mobile Robots*, (2011).

[10] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger, 'Using the context-enhanced additive heuristic for temporal and numeric planning', in *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, (2009).

[11] David E. Foulser, Ming Li, and Qiang Yang, 'Theory and algorithms for plan merging', *Artificial Intelligence*, **57**(23), 143 – 181, (1992).

[12] C. Harris and R. Dearden, 'Contingency planning for long-duration auv missions', in *2012 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pp. 1–6, (Sept 2012).

[13] Malte Helmert, 'The fast downward planning system', *Journal of Artificial Intelligence Research*, **26**, 191–246, (2006).

[14] Subbarao Kambhampati, 'Refinement planning as a unifying framework for plan synthesis', *Artificial Intelligence Magazine*, **18**(2), 67–97, (1997).

[15] Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang, 'Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning', *Artificial Intelligence*, **76**, 167–238, (1995).

[16] Wing-Yue Geoffrey Louie, Tiago Stegun Vaquero, Goldie Nejat, and J. Christopher Beck, 'An autonomous assistive robot for planning, scheduling and facilitating multi-user activities', in *Proc. of 2014 IEEE Int. Conf. on Robotics and Automation (ICRA)*, (2014).

[17] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen, 'T-rex: A model-based architecture for auv control', *3rd Workshop on Planning and Plan Execution for Real-World Systems*, **2007**, (2007).

[18] Lenka Mudrova, Bruno Lacerda, and Nick Hawes, 'An integrated control framework for long-term autonomy in mobile service robots', in *2015 European Conference on Mobile Robots (ECMR)*. IEEE, (2015).

[19] Bernhard Nebel and Jana Koehler, 'Plan reuse versus plan generation: A theoretical and empirical analysis', *Artificial Intelligence*, **76**, 427–454, (1995).

[20] Filip Dvořák, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab, 'A flexible anml actor and planner in robotics', in *ICAPS PlanRob WorkShop*, Portsmouth, USA, (June 2014).

[21] Masood Feyzbakhsh Rankooh and Gholamreza Ghassem-Sani, 'New encoding methods for sat-based temporal planning', 110–117, (2013).

[22] Maurizio Di Rocco, Federico Pecora, and Alessandro Saffiotti, 'When robots are late: Configuration planning for multiple robots with dynamic goals.', in *Proc. of 2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, (2013).

[23] Earl D. Sacerdoti, 'The nonlinear nature of plans', in *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'75, pp. 206–214, (1975).

[24] Reid G. Simmons and Håkan L. S. Younes, 'VHPOP: versatile heuristic partial order planner', *CoRR*, **abs/1106.4868**, (2011).

[25] D. E. Smith, J. Frank, and W. Cushing, 'The anml language', in *ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, (2008).

[26] A. Tate, 'Generating project networks', in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, IJCAI'77, pp. 888–893, (1977).

[27] Ioannis Tsamardinos, Martha E. Pollack, and John F. Horty, 'Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches', in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pp. 264–272, (2000).

[28] Roman Van Der Krogt and Mathijs De Weerdt, 'Plan repair as an extension of planning', in *In Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05*, pp. 161–170, (2005).

[29] Manuela M. Veloso, Joydeep Biswas, Brian Coltin, Stephanie Rosenthal, Thomas Kollar, Cetin Mericli, Mehdi Samadi, Susana Brandao, and Rodrigo Ventura, 'Cobots: Collaborative robots servicing multi-floor buildings.', in *Proc. of 2012 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, (2012).

[30] Vincent Vidal, 'YAHSP2: Keep it simple, stupid', in *Proceedings of the 7th International Planning Competition (IPC-2011)*, pp. 83–90, Freiburg, Germany, (June 2011).

[31] Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi, 'Adaptive K-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning', in *Proceedings of the 3rd Symposium on Combinatorial Search (SOCS-2010)*, pp. 100–107, Stone Mountain, GA, USA, (July 2010). AAAI Press.

[32] Daniel S. Weld, 'An introduction to least commitment planning', *AI magazine*, **15**(4), 27, (1994).

[33] Qiang Yang, *Intelligent Planning: A Decomposition and Abstraction Based Approach*, Springer-Verlag, London, UK, 1997.