28th International Conference on
Automated Planning and Scheduling

June 24–29, 2018, Delft, the Netherlands

Hierarchical Planning 2018

Proceedings of the 1st ICAPS Workshop on

Hierarchical Planning

Edited by:

Pascal Bercher, Daniel Höller, Susanne Biundo, and Ron Alford

Organizing Committee

Pascal Bercher	Ulm University, Germany
Daniel Höller	Ulm University, Germany
Susanne Biundo	Ulm University, Germany
Ron Alford	The MITRE Corporation, McLean, Virginia, USA

Program Committee

Ron Alford	The MITRE Corporation, McLean, Virginia, USA
Roman Barták	Charles University, Prague, Czech Republic
Gregor Behnke	Ulm University, Germany
Pascal Bercher	Ulm University, Germany
Susanne Biundo	Ulm University, Germany
Kutluhan Erol	İzmir University of Economics, Turkey
Robert P. Goldman	SIFT, LLC, Minneapolis, USA
Daniel Höller	Ulm University, Germany
Ugur Kuter	SIFT, LLC, Minneapolis, USA
Felix Richter	Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Stuttgart, Germany
Mak Roberts	U.S. Naval Research Laboratory, Washington, DC, USA
Vikas Shivashankar	Amazon Robotics, North Reading, Massachusetts, USA

Preface

The motivation for using hierarchical planning formalisms is manifold. It ranges from an explicit and predefined guidance of the plan generation process and the ability to represent complex problem solving and behavior patterns to the option of having different abstraction layers when communicating with a human user or when planning cooperatively. This led to a large set of different hierarchical formalisms and systems. With this workshop, we want to bring together scientists working on any aspect related to hierarchical planning to exchange ideas and foster cooperation.

Hierarchies induce fundamental differences from classical, non-hierarchical planning, creating distinct computational properties and requiring separate algorithms for plan generation, plan verification, plan repair, and practical applications. Many of these aspects of hierarchical planning are still unexplored.

This wide range of important yet insufficiently solved problems is reflected in the topics presented in this proceedings. Though the main focus lies on the development of planning systems, these tackle quite different classes of hierarchical problems and use several solving techniques. It includes work on real-time planning, planning with task insertion, distributed planning, and extensions of formalisms to enable real-world application. Beside solvers, the presented work includes techniques for the plan repair problem and discussions of the application in real-world problems.

Pascal, Daniel, Susanne, and Ron
Workshop Organizers,
June 2018

Table of Contents

A Depth-Balanced Approach to Decompositional Planning for Problems where Hierarchical Depth is Requested

David R. Winer[1] and **Rogelio E. Cardona-Rivera**[1,2]
[1]School of Computing
[2]Entertainment Arts and Engineering Program
University of Utah
Salt Lake City, UT, USA
{drwiner, rogelio}@cs.utah.edu

Abstract

Hybrid planning with task insertion for solving classical planning problems, or decompositional planning, combines partial-order causal link planning with hierarchical task networks, where steps in the plan may represent composite (i.e., compound) actions that are decomposable into sub-steps using hierarchical knowledge. We have designed a planning algorithm that responds to a request for maximizing the hierarchical depth of plans while minimizing the plan length. In some applications, plans that adhere to hierarchical constraints are preferred over other valid plans. One of the main obstacles of this challenge is to incentivize the planner to insert composite actions while avoiding excessive search on the depth attribute. We introduce plan scoring heuristics that avoid over-discounting and under-discounting depth using a novel way to measure plan depth. We evaluate these heuristics on test problems and demonstrate that we can generate deep, low-cost solutions to planning problems while avoiding excessive search.

Hybrid planning is a plan-space planning paradigm that combines partial-order causal link reasoning (Weld 1994) with hierarchical knowledge (Erol, Hendler, and Nau 1994) in order to solve a hybrid planning problem (the refinement of an initial partial plan into a plan with no flaws). While there exist several variants of hybrid planning (e.g. Young, Pollack, and Moore 1994, Lee-Urban 2012, Bechon et al. 2014), all variants afford some representation of task hierarchies through two kinds of tasks (i.e. steps): primitive and composite. The former are similar to steps in partial-order causal link (POCL) planning. The latter are drawn from hierarchical task network (HTN) planning (but also contain preconditions and effects as in POCL planning); they represent abstract tasks involving several more-primitive steps. Whereas a primitive step that has been added to a plan can be directly executed (assuming its preconditions hold), composite steps that have been added are not directly executable; a more primitive sub-plan for the composite step must be found that depends on the composite's preconditions and that achieves the composite's effects. Such a sub-plan may be input to a hybrid planner through a **decomposition method**.

Our planning applications make a non-standard *depth request* for hybrid planners. The request is that the planner maximizes the ratio of hierarchical depth (number of decomposition methods) to plan length (number of primitive tasks) of generated plans. The number of decomposition methods it uses to refine the initial plan is an integral part of the planning problem's solution. Composite tasks are not wholly substituted for sub-plans that decompose them, but rather kept around to identify the hierarchical structure inherent in the plan. The underlying assumption is that the high-level structure of the plan (identified through the decomposition methods) is implicitly meaningful or useful for the planning agent. For example, in planning-based natural language generation (Garoufi 2014), plans may be preferred if they follow recognizable discourse patterns, which may be computed from data-driven observations and operationalized as hierarchical knowledge. Another example is the case of planning-based narrative generation (Young et al. 2013), wherein plans may be preferred if they follow normative narrative structure, often analytically identified as containing hierarchical segments (Prince 2003; Bordwell, Thompson, and Smith 1997).

Typically, hybrid planners will insert a composite task, rather than a primitive task, if the composite task is explicitly needed because it has some *primary effect*: an effect that none of the tasks in its decompositional refinement can establish on its own (Kambhampati, Mali, and Srivastava 1998). A primary effect can characterize something that is more than the sum of its parts; for example, an argument is made by refuting facts, establishing background, referring to evidence, and making a conclusion, but none of these items are sufficient on its own. In contrast, in the classic *travel planning domain*, a goal to be located at a destination is achieved by a primitive task of exiting a plane that is at said destination; thus, a composite task – e.g. travel-by-plane – is not inserted into a plan unless it is estimated to save the planner time and effort for repairing the same goal condition. Given that heuristics tend to underestimate effort saved, and composite tasks add more to the plan length than primitive tasks, a planner using a best-first search is going to evaluate a repair with a primitive task as cheaper than a plan that makes the same repair with a composite task. We address the problem of fulfilling the depth request in hybrid domains where composite tasks may not have primary effects.

The decision point we focus on is which task to insert to repair *open conditions*, which (in POCL planning) are flaws in the plan generated when a step has unsatisfied (i.e. open) preconditions and repaired by ensuring that the preconditions are established by some earlier action; prior work has focused on selecting a decomposition method for a composite task. State-of-the-art heuristics that exploit task decomposition (e.g. Minimal Modification Effort by Bercher, Keen, and Biundo 2014) are not of use here. These heuristics bias the planner to refine the initial plan to primitive tasks in the least number of refinements (as they should do to generate the shortest plans). In our methodology, each composite task is fully decomposed before it is inserted into a plan. Thus, the hierarchical depth of a composite task is known at the moment of insertion, and its entire decomposition refinement tree (its sub-plan) is inserted into the plan. The main question we ask is *how should the planner best add hierarchical depth to the plan during task insertion to fulfill the depth request*? Our approach is to have the planner leverage the known depth of the decomposition tree of inserted composite tasks. Incentivizing the insertion of composite tasks comes with a tradeoff: the planner must search a much larger space of plans and significantly reduce efficiency. The key problem we address is how to have the planner incentivize inserting composite tasks without a significant tradeoff to efficiency.

Contributions We introduce a plan selection function that reifies the tradeoff between inserting deep and shallow tasks within hybrid planning with task insertion. This function incentivizes selecting plans with composite tasks on the search frontier and depends on a novel way to measure plan depth. We evaluated our function's effectiveness on the basis of runtime performance and solution depth on test problems, and demonstrate we can find deep, low-cost solutions while avoiding brute-force search on hierarchical depth.

Related Work

Our planning task is an instance of hybrid planning with task insertion. Terminologically, we use the term "decompositional planning" to describe what our planner is doing: solving a problem given in terms of goal literals as in classical planning, but wherein abstract tasks can be inserted as well (i.e. there are no initial abstract tasks, but rather solving proceeds from an initial dummy plan given by the initial conditions and the goal literals). While we recognize that there are planning variants that differ terminologically and semantically (e.g. "hybrid planning" by Kambhampati, Mali, and Srivastava 1998), reconciling all variants is beyond the scope of this paper. In some variants, all (causally necessitated) tasks are specified as part of a decomposition method (Elkawkagy et al. 2012; Bercher, Keen, and Biundo 2014) and therefore task insertion is unneeded (or at least not allowed in the context of the problem being addressed). It is not always possible to specify all tasks in a decomposition method: some open condition of a task may not have a supplier within the same decompositional hierarchy. Thus, task insertion is allowed to repair open conditions that are left open as in Kambhampati, Mali, and Srivastava (1998) and Geier and Bercher (2011). Inserted tasks can either be primitive or composite.

One hybrid planning approach by Elkawkagy et al. (2012) which does not allow task insertion is to compile a Task Decomposition Graph (TDG) composed of edges connecting tasks (primitive or composite) to decomposition methods and vice versa. The TDG is used to guide the planner to shorter plans from an initial task representing the initial partial plan so that the solution is a refinement of the initial task (Bercher, Keen, and Biundo 2014). The criteria for a solution is that all composite tasks are decomposed into primitive tasks, all tasks are fully grounded, and all open conditions are repaired by other tasks in the plan. In our approach, a composite task is fully decomposed and grounded before the entire sub-tree is inserted into a plan to repair an open condition. The decompositions are performed in a precaching stage where a max number of decompositional refinements (i.e. step height) is used to cutoff search. The decision points we consider in this work is not which decomposition method to select to decompose a composite task in a plan, but rather which fully ground and decomposed composite task tree to insert to repair an open condition in a plan. Because the sub-tasks in the tree may have open conditions, the TDG will under-estimate the number of decompositions in the plan because inserted tasks may also be composite.

A planning domain and problem can (intentionally or inadvertently) require hierarchical depth in the plan to solve the problem. DPOCL-T (Jhala and Young 2010), a variant of DPOCL (Young, Pollack, and Moore 1994) for scheduling camera shots in narrative generation, is tested using a domain that is engineered to promote hierarchical depth by leveraging primary effects. At each "tier" in a multi-level hierarchy, high-level operators have preconditions that can only be fulfilled by other high-level operators at the same level. Thus, a problem whose goal is a primary effect of a high-level action will require the planner to create a high-level plan. In a similar vein, HiPOP (Bechon et al. 2014) uses stages to create plans with composite steps. In the first round, only composite steps are applicable and must be used for as long as possible without expanding them until no composite steps can be used to satisfy preconditions of other steps (or the goal conditions). If no solution can be found, then HiPOP will never proceed to the expansion round.

The forward state-space hybrid planner UPS (To, Langley, and Choi 2015) favors states produced by composite steps when that state satisfies elements of the goal formula. Its heuristic counts the number of unmatched goal elements when selecting steps for expansion, greedily selecting composite steps. We suspect that UPS will excessively search through a large space of composite tasks because of this greedy heuristic. In future work, we plan to compare UPS to our own approach.

The Language of Decompositional Planning

The formal model of decompositional planning we adopt is taken from DPOCL, a planning system previously developed by Young, Pollack, and Moore (1994). DPOCL builds on POCL planning, which searches in the space of plans to find a partial plan (i.e. a set of steps S, a set of partial ordering relations O over S, and a set of causal links L) with no *flaws*; all preconditions must be satisfied (no condition may remain *open*) and no causal links may be *threatened* (i.e. it should not be possible for any step to be ordered such that it potentially undoes a causal link's protected literal). Open conditions are repaired through *adding* or *reusing* a plan step and threatened causal links are repaired by *promoting* or *demoting* the offending step to come after or before (respectively) the threatened link. For a more thorough introduction to POCL planning, we refer the reader to Weld (1994).

DPOCL has two kinds of steps. A *primitive step* is as in POCL planning. A *composite step* is a step that is decomposable into a partial-plan, called a *sub-plan*. Each composite step is a composite operator type paired with a set of bindings over operator parameters. In this paper, composite steps are associated with a single decomposition method and a set of bindings over decomposition parameters (i.e. a composite operator is rewritten with a specific decomposition method). A step in the sub-plan of a composite step is a *sub-step*. The decomposition specifies constraints over partially defined sub-steps that are useful in our application. A *decompositional link* relates a composite step to a substep. The *height* of a composite step is the longest path of decomposition links. Thus a decompositional plan is represented by a tuple $\langle S, O, L, D\rangle$ where D is a set of decompositional links. The goal of our planner is to generate a DPOCL plan that solves an input decompositional planning problem and maximizes the ratio of decomposition links to primitive steps.

Motivating Problem

Our approach is broadly motivated by the goal of fostering successful human-computer communication. Humans implicitly use grammar to understand and recognize the meaning of speakers (Sperber and Wilson 1987). Strictly, a grammar defines what is a well-defined sequence. In human communication, a grammar may be informal and describe what is an easily recognized pattern of utterances. In human-in-the-loop planning, a planning agent can pose queries to a human operator in the decision of how to continue the planning process (Roth et al. 2004; Schirner et al. 2013). These systems can leverage the way human communicators structure information to more easily recognize user intent and select plans that are not cognitively demanding to parse.

The specific application for a decompositional planner we focus on is the task of directing film. A film director controls various details related to how agents (i.e. character actors) perform actions in an environment and how camera shots should convey those details. Film directors plan out visual details across shots to build up a hierarchically-structured editing pattern. An editing pattern is composed of camera shots, and the sequence that shots cut from one to the next (i.e. transition) affects the way that viewers focus on events. The general principle is that good cuts have matching visual details across shots. The more that shots conform to an editing pattern, the better the aesthetic quality of the sequence. In computer graphics research, camera control systems find camera sequences that best adhere to a grammar of film (He, Cohen, and Salesin 1996; Christianson et al. 1996; Christie, Olivier, and Normand 2008).

Formulated as a decompositional planning task, the film director (i.e. the planner) selects the content and style of camera shots (primitive tasks) to compose a scene. Each task has preconditions and effects: preconditions correspond to the necessary world conditions for the actor to perform in the world to create the camera shot content, and effects correspond to conditions that change in the world state as a result of the actor's performance. The more that camera shots adhere to editing patterns (decomposition methods), the better the quality of the resulting film. The decomposition methods impose constraints on camera shot attributes (e.g. scale, angle) and character actions (e.g. orientation, timing) that give rise to good editing transitions. Thus, the quality of the solution is based on the degree that camera shots adhere to editing patterns and not entirely on the length of the plan. Figure 2 presents two short editing sequences: sequence *A* has good individual shots but does not adhere to a pattern, whereas sequence *B* adheres to an editing pattern and results in matched visual details across shots.

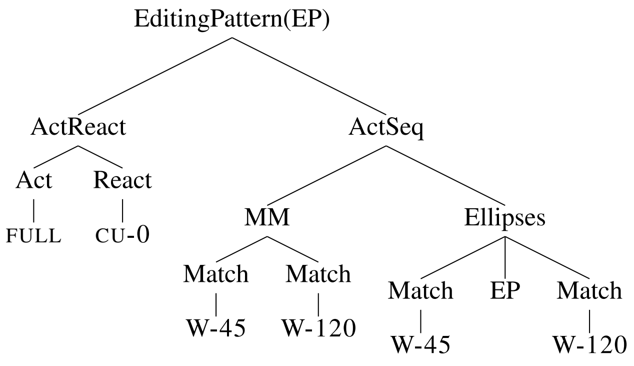

Figure 1: Example Film Editing Grammar

The "film directing" decompositional planning task motivates the depth request:

$$\textsc{DepthRequest} = \max_{\pi} \frac{|\{s \in S(\pi) : height(s) > 0\}|}{|\{s \in S(\pi) : height(s) = 0\}|}$$

maximize the ratio of decomposition methods to primitive plan steps in a solution to a decompositional planning problem. This ratio corresponds to the percentage of camera shots that adhere to an editing pattern. For each task that the planner inserts to repair an open condition, the planner must decide whether to add hierarchical depth to the plan's structure, and if so, how much. Figure 1 shows an example grammar tree for film editing. The leaves of the tree are camera shots, with the exception of EP where another editing pattern can be refined into camera shots. To repair an open condition, the planner decides whether to insert a single camera shot (primitive task) or an editing pattern of some depth (composite task), and each may introduce new open condition flaws to establish the prerequisite world conditions.

Approach

The approach has two stages: *first*, a composite step is compiled for every valid sub-tree of a task decomposition graph (a sub-tree is valid just when its leaves are primitive tasks) up to positive non-zero tree height cutoff $h_{\max}$. These composite tasks are pre-cached in a similar manner to mod-

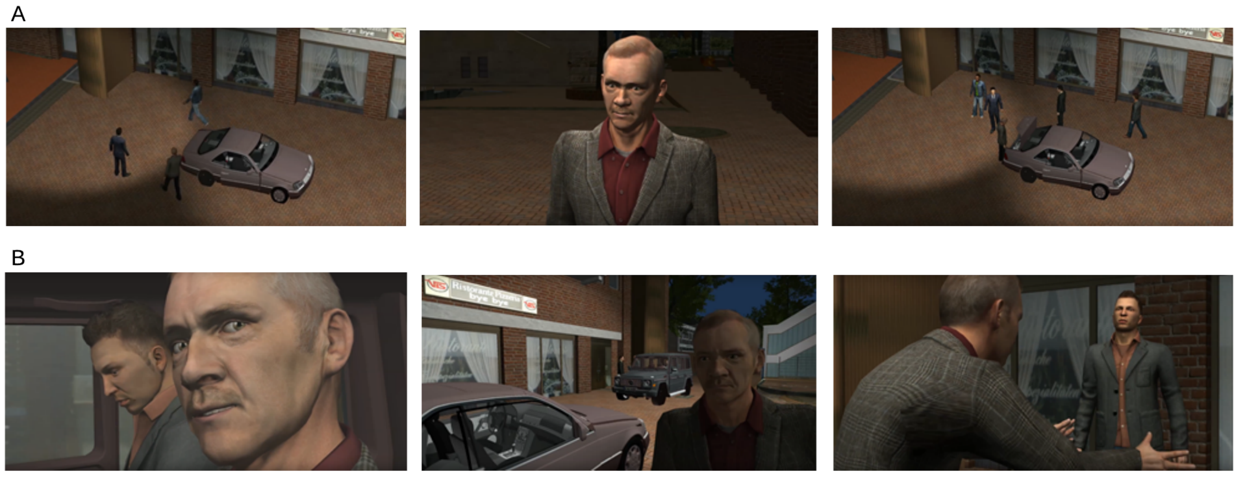

Figure 2: Sequences **A** and **B** are camera shots sampled from different versions of the same film created by a film director. **A** has good individual shots but sequences in the film are discontinuous and do not follow editing patterns. **B** contains sequences developed to adhere to editing patterns. This results in matched visual details across shots and improved focus on events.

ern POCL planners (e.g. VHPOP, developed by Younes and Simmons (2003) pre-caches all possible ground instances of problem operators as steps). After grounding the primitive operators (defined to be $h = 0$), grounding continues with composite operators in a bottom-up manner. The first round of composite operators to be grounded ($h = 1$) can only use ground elements and steps of height 0. Inductively, composite steps that have been grounded at height h can serve as sub-steps for grounding composite operators at height $h+1$ up to h_{max} (exclusive). Thus, a **composite step** is defined relative to the height where it has been grounded.

Definition 1 (Composite Step) *Represents an instance of a composite operator λ_c at height h. It is a tuple $\lambda_c^G = \langle \lambda_c, B, \pi_u, h\rangle$, where B is a set of consistent bindings for the variables in $V \in \lambda_c$ and $\pi_u = \langle \mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{D}\rangle$ is a sub-plan for a decomposition of λ_c such that $\mathcal{S}$ contains at least one step whose height is exactly $h - 1$ and no steps with height greater than $h - 1$.*

Our method is a dynamic programming approach to recursive HTN planning that can be used when the maximum depth is known *a priori*. Composite operators are grounded for every applicable decomposition method (at each height $h = 1$ to h_{max}). The decomposition method is used to calculate the sub-plan and variable bindings that are defined in a composite step. This dynamic programming method was developed by Winer and Young (2017).

The *second* stage is to solve the planning problem using pre-cached steps. The algorithm follows a classic POP with the addition that adding composite steps leads to the insertion of its pre-cached sub-plan and the decompositional links that point the composite step to its sub-steps.

$$\begin{tikzcd}
 & & & & 0 \arrow[dlll, "a"'] \arrow[d, "a"] \arrow[drr, "a"] & & \\
 & 00 \arrow[dl, "r"'] \arrow[dr, "a"] & & & 01 \arrow[dl, "r"'] \arrow[d, "a"] \arrow[dr, "a"] & & 02 \arrow[d, "a"] \\
000 & & 001 & 010 & 011 \arrow[dl, "p"'] \arrow[dr, "d"] & 012 & 020 \\
 & & & 0110 & & 0111 &
\end{tikzcd}$$

Figure 3: A plan search tree. Nodes are plans and edges are refinements to the plan. The edge label represents the repair method; one of $a$dd, $r$euse, $p$romotion, and $d$emotion.

Plan Search Tree and Plan Depth

Our novel notion of **plan depth** is based on the reasons that steps are added to the plan. It depends on two structures: decomposition links (as defined earlier) and **add-repair arcs**. To define add-repair arcs, it is useful to talk about the planning process by way of its search tree. The **plan search tree** represents the search space of plans and their refinements. A vertex is a pair (π, F_π) where π is a plan and F_π is a set of flaws in π. Edges are of the form $(\pi, F_\pi) \xrightarrow{(f,\rho)} (\pi', F_{\pi'})$ where $f \in F_\pi$ is the flaw selected for π (the parent) and π' (the descendant) is the result of repairing flaw f with refinement ρ (one of $\{\text{add}, \text{reuse}, \text{promote}, \text{demote}\}$). $F_{\pi'} =$

$F_\pi - f \cup F_\rho$ where F_ρ are flaws detected in π' after ρ. A **path** is a sequence of nodes and edges connecting a vertex with a descendant.

Let q be a path on a plan search tree, and $\pi' = \langle S, O, L, D \rangle$ be a plan on a leaf vertex. An **add-repair arc** exists between two steps $s_i, s_j \in S$ from s_j to s_i just when there exists a causal link of the form $s_i \xrightarrow{p} s_j \in L$ such that s_i was added to S through an edge $(\pi, F_\pi) \xrightarrow{(\langle s_i, p\rangle,\ \text{add})} (\pi', F_{\pi'})$ (i.e. an open condition p for step s_i repaired through adding a step) for some parent π.

Definition 2 (Deep Path and Plan H-depth) *A* **deep path** *δ on a plan π is a traversal of decomposition links and add-repair arcs in the plan; the length of a deep path $len(\delta)$ is the number of decomposition links in the traversal. The* **H-depth** *of π is the longest deep path on π.*

When steps are reused to repair an open condition, no plan H-depth is added. H-depth is recalculated only after adding steps and requires only constant time to track.

Algorithm

The *Ground Decompositional Partial Order Planning algorithm* (GDPOP) presented in Algorithm 1 is a propositional POCL planner that uses composite steps whose decomposition refinements are already performed. GDPOP takes as input a set of ground primitive steps Λ_p^G, a set of ground composite steps Λ_c^G, the plan-space initial plan, a **candidate map**, which maps every open condition of every step $\langle s_{\text{need}}, p \rangle$ to every step with an effect p), and a **threat map**, which maps every open condition of every step $\langle s_{\text{need}}, p \rangle$ to every step with an effect $\neg p$. Initially, the plan has depth 0 and steps are assigned a depth of 0. When a composite step is *added* to repair an open condition (*a la* Algorithm 2), sub-steps are *inserted* and depth is propagated (*a la* Algorithm 3).

At each iteration, GDPOP identifies all flaws in the plan under consideration, selects a flaw to fix, and adds to the search fringe all the plans that represent every way in which the flaw could be repaired. Selecting which flaw to repair affects the order that plans are visited in the search space (Younes and Simmons 2003); we adopted a simplified version which is

1. open conditions that are static (are unchangeable given the problem's operators)
2. threatened causal link flaws
3. open conditions that hold initially
4. most unsafe open conditions first (with at least one potential risk detected using the threat map)
5. open conditions with at least 1 candidate for reuse (detected using the candidate map, sorted by random hash code)
6. open conditions with no option for reuse

A risk s_{risk} in a plan π for an open condition of the form $\langle s_{need}, pre \rangle$ is a step that may undo the open condition because $s_{risk} \in T_{\text{MAP}}[pre]$ and $\nexists s_{need} \prec s_{risk} \in O(\pi)$. The number of risks for an open condition is set at the moment it is created (we did not update the number of risks when inserting new steps or remove risks when a step is no longer a risk). A candidate s_{cndt} in a plan for an open condition is a step in $C_{\text{MAP}}[pre]$ that can be ordered before s_{need}. Similarly, no update is made to update the number of candidates after the open condition is created.

Algorithm 1 `GDPOP`

Input: Candidate map C_{MAP}, threat map T_{MAP}, initial plan π_0 with steps s_0, s_∞ (all of depth 0), a function $\mathcal{F}$ returning flaws F for a plan, and a plan selection function $\mathcal{E}$.
Output: A consistent plan with no flaws or failure.

```
 1: $\mathcal{OL} := openList$.push($\pi_0$)
 2: while $\mathcal{OL}$ not empty do
 3:     $\pi = \langle S, O, L, D \rangle := \arg\max_{\pi \in \mathcal{OL}} \mathcal{E}(\pi)$
 4:     if cycle in $O$, then skip
 5:     end if
 6:     $F := \mathcal{F}(\pi, T_{\text{MAP}})$, return $\pi$ if $|F| = 0$
 7:     $f := Flaw\text{-}Select(F, C_{\text{MAP}}, T_{\text{MAP}})$
 8:     if $f$ is an open condition then
 9:         $\mathcal{OL}$.push(Add($\pi, f, C_{\text{MAP}}$))
10:         $\mathcal{OL}$.push($Reuse(\pi, f, C_{\text{MAP}})$)
11:     else if $f$ is a threatened causal link then
12:         $\mathcal{OL}$.push($\pi_{\text{promote}}$ and $\pi_{\text{demote}}$)
13:     end if
14: end while
15: return FAIL
```

Algorithm 2 `Add`

Input: π, $\langle s_{need}, p \rangle$, C_{MAP};
Output: Expanded π

```
1: $\Pi = \varnothing$
2: for each step $\lambda$ in $C_{\text{MAP}}[p]$ do
3:     $s_{new} := \lambda.clone()$; $s_{new}.depth = s_{need}.depth$
4:     $\pi' := \pi.clone()$
5:     Insert($\pi', s_{new}$); $Repair(\pi', s_{new}, s_{need}, p)$
6:     $\Pi.add(\pi')$
7: end for
8: return $\Pi$
```

We detail the plan refinement operations that use our novel constructs (i.e. Algorithm 2, Algorithm 3) and leave un-specified the plan refinement operations that are taken from standard POCL planning (i.e. $Reuse$ in Algorithm 1, $Repair$ in Algorithm 2, and the calculation of plans π_{promote} and π_{demote} that promote and demote steps that threaten causal links, respectively).

A Depth-Balancing Plan Selection Heuristic

The evaluation of a plan, which orders plans in the search frontier (open list) in a best-first plan-space search, is defined as $\mathcal{E}^0(\pi) = g(\pi) + h(\pi)$ where $g(\pi)$ denotes the plan **cost**, i.e. the number of steps in the plan, and $h(\pi)$ denotes the **heuristic** value, an estimate of the number of steps which need to be inserted into the plan to solve the problem.

We adopt VHPOP's *additive-reuse heuristic* function for calculating the heuristic value. The function recursively simulates new open conditions created by inserting actions to make repairs until all open conditions hold initially. Nota-

Algorithm 3 `Insert`

```
Input: $\pi, s_{new}$
 1: Add $s_{new}$ to $\pi$
 2: if $height(s_{new}) > 0$ then
 3:     for each sub-step $s$ in SUB-PLAN($s_{new}$) do
 4:         Add decompositional link $s_{new} \rightharpoondown s$ to $\pi$
 5:         $s.depth = s_{new}.depth + 1$
 6:         if $s.depth > \pi.depth$ then
 7:             $\pi.depth = s.depth$
 8:         end if
 9:         Insert($\pi, s$)
10:     end for
11: end if
```

tion: C_{MAP} is a candidate map, which maps each precondition to the set of actions that can repair it, and $\mathcal{OC}$ returns the set of open conditions in a plan.

$$h^r_{add}(\pi) = \sum_{\xrightarrow{q} a_i \in \mathcal{OC}(\pi)} \begin{cases} 0 & \text{if } \exists a_j \in S \cap C_{\text{MAP}}[q] \\ & \text{and } a_i \prec a_j \notin O \\ h_{add}(q) & \text{otherwise} \end{cases}$$

$$h_{add}(q) = \begin{cases} 0 & \text{if } q \text{ holds initially} \\ \min_{a \in C_{\text{MAP}}[q]} h_{add}(a) & \text{if } C_{\text{MAP}}[q] \neq \varnothing \\ \infty & \text{otherwise} \end{cases}$$

$$h_{add}(a) = 1 + \sum_{p \in pre(a)} h_{add}(p)$$

In addition to cost g and heuristic h, we introduce a value d that corresponds to H-depth (as in Definition 2). We experimented with several ways to discount this value:

1. $\mathcal{E}^1(\pi) = g(\pi) + h^r_{add}(\pi) - d(\pi)$
2. $\mathcal{E}^2(\pi) = g(\pi) + h^r_{add}(\pi) - \log_2(d(\pi) + 1)$
3. $\mathcal{E}^3(\pi) = \frac{g(\pi)}{1+\log_2(d(\pi)+1)} + h^r_{add}(\pi)$
4. $\mathcal{E}^4(\pi) = g(\pi) + h^r_{add}(\pi) - |\{s \in S(\pi) : height(s) > 0\}|$
5. $\mathcal{E}^5(\pi) = \frac{g(\pi)^2}{1+|\{s \in S(\pi) : height(s) > 0\}|} + h^r_{add}(\pi)$
6. $\mathcal{E}^6(\pi) = |\{s \in S(\pi) : height(s) > 0\}| + h^r_{add}(\pi)$

In $\mathcal{E}^2$ and $\mathcal{E}^3$, the logarithm helps diminish the extent to which plan H-depth drives the score. The gist of these functions is that composite steps are considered less valuable as repairs the deeper the plan gets, and therefore protects against over-incentivizing depth. $\mathcal{E}^2$ subtracts depth and $\mathcal{E}^3$ divides by it; we compared them to evaluate the sensitivity of the search to how depth is factored into the score. We hypothesized that $\mathcal{E}^3$ would find the best depth-request ratios.

Preliminary Evaluation 1

We developed and compared GDPOP planning with different plan selection heuristics on a generic domain. Our focus is on the performance of the heuristics on each problem.

Table 1: A comparison of plan quality across experiment conditions. All values are averages produced across problems; $g(\pi)_\text{m}$ is the mean cost of solutions, S is the number of solutions out of 320, $d(\pi)_\text{m}$ is the mean depth of solutions, and $d_\text{max}(\pi)_\text{m}$ is the mean maximum depth of solutions, where the maximum is defined over solutions for a given planning problem.

	$g(\pi)_\text{m}$	S	$d(\pi)_\text{m}$	$d_\text{max}(\pi)_\text{m}$
g_{PO}	6.89	320	0.00	0.00
$g_{Insrt}\ \mathcal{E}^0$	5.89	320	0.61	0.88
$g_{Insrt}\ \mathcal{E}^1$	6.58	320	0.30	1.13
$g_{Insrt}\ \mathcal{E}^3$	9.58	320	1.36	2.75
$g_{Add}\ \mathcal{E}^0$	3.34	287	1.01	2.38
$g_{Add}\ \mathcal{E}^1$	2.79	152	2.10	3.71
$g_{Add}\ \mathcal{E}^2$	3.45	290	1.69	2.88
$g_{Add}\ \mathcal{E}^3$	**5.91**	**281**	**1.98**	**4.38**

In addition to $\mathcal{E}^1, \mathcal{E}^2, \mathcal{E}^3$ as candidate depth-balancing plan selection functions, we also considered the following algorithm variants:

- g_{PO} (primitive-only) indicates that only primitive steps are provided as input.
- g_{Insrt} (with composite steps) adds 1 to the cost for every *Insert* operation.
- g_{Add} (with composite steps) adds 1 to the cost for every *Add* operation (and therefore sub-steps are inserted for free).

Methods Python was used to prototype the idea and hypothesis[1]. Subsequently, C# was used as part of the port to the *Unity Game Engine* for film directing, and demonstrates the run time efficiency more realistically. We ran the prototype implementation of GDPOP on sample problems which vary in number of objects, initial conditions, or goal conditions. The composite operators in the domain are based on filming the classic travel domain. Eight (8) problems were constructed that averaged 2 agents, 2.125 vehicles, 2.5 locations, 1.625 goal conditions. First, the steps are pre-cached. On average, the problems included 45.25 compiled primitive steps (λ^G_p), and 201 compiled composite steps (λ^G_c). The maximum step height is 2. The largest problem (#8) has 4 agents, 4 locations, 2 vehicles, and 2 goal conditions. The planner is run on a 64-bit Windows 7 machine with an Intel i7-3770 CPU at 3.40 GHz and 16 GB of RAM. For each experimental condition, the planner was run until 40 solutions were generated or 400 seconds had elapsed. The conditions of the experiment are g_{PO}, $g_{Insrt}\ \mathcal{E}^i$, and $g_{Add}\ \mathcal{E}^i$ for $i = 0 - 3$; a total of 9 conditions.

Results We analyzed the performance of the planner on the basis of runtime and nodes expanded. In general, the primitive-only condition is fastest but expands far more nodes (as it should given that composite steps can add many primitive steps in a single node). The g_{Insrt} and $g_{Insrt}\ \mathcal{E}^1$

[1]`https://github.com/drwiner/PyDPOCL`

Table 2: Plan Evaluation Results (Preliminary Evaluation 2). *Legend*: Ev: plan scoring function, Heu: heuristic, AR: "AddReuse", OC: "number of open conditions", RT: runtime (milliseconds), Op: "number of nodes opened", Exp: "number of nodes expanded", Co: cost, D: number of decomposition methods, R: depth request ratio, S: number of problems solved out of 8. The "zero" heuristic was also run for each evaluation, but not included in cases where no problems are solved. We also ran a breadth-first search which did not solve any problems.

Ev	Heu	RT	Op	Exp	Co	D	R	S
E0	AR	734	1744	185	4.5	0.3	0.4	8
E0	OC	346	1087	180	3.3	0	0	7
E1	**AR**	**697**	**1557**	**165**	**6.3**	**1.1**	**1.5**	**8**
E1	OC	334	1044	175	5.3	1	1.4	7
E2	**AR**	**690**	**1526**	**161**	**6.3**	**1.1**	**1.5**	**8**
E2	OC	332	1012	168	5.3	1	1.4	7
E3	**AR**	**123**	**360**	**41.4**	**6.4**	**1.1**	**1.5**	**8**
E3	**OC**	**191**	**730**	**87**	**6.3**	**1.1**	**1.5**	**8**
E4	AR	720	1744	185	4.6	0.3	0.4	8
E4	OC	333	1087	180	3.3	0	0	7
E5	AR	29	77	12	5	1	0.6	3
E5	OC	401	1027	199	9	2	0.4	2
E6	AR	47	166	21	4.2	0	0	6
E6	OC	44	232	28	4.3	0	0	8
E6	Zero	1509	5430	966	3	0	0	5
DFS	Zero	11	61	13	8	1	0.02	1

conditions perform very similarly, expanding less nodes than PO but more than the g_{Add} conditions. The results suggest that the g_{Add} cost function with the $\mathcal{E}^3$ scoring function expands the fewest nodes on average.

Table 1 shows the quality of solutions for each experimental condition across planning problems. Across the experimental conditions, we observe that cost is weakly sacrificed for plan H-depth, and that $\mathcal{E}^3$ performs best for finding the deepest solutions. Although g_{Add} $\mathcal{E}^1$ appears to find the best average depth, it does not find solutions on one of the planning problems in the allotted time (problem 8) and generally struggled on other problems, whereas g_{Add} $\mathcal{E}^2$ and $\mathcal{E}^3$ found 40 solutions on this problem before the time cutoff.

Preliminary Evaluation 2

The first experiment sought to evaluate plan selection criteria that would promote hierarchical depth. However, the depth request ratio was not measured. We ran a new experiment to evaluate the average depth request ratio, to compare against baseline heuristics, and to compare against other selection functions that are potentially depth-balancing. The GDPOP algorithm is reimplemented in C#[2] as part of the film directing application. Table 2 shows performance and quality averages for the first solution across the 8 problems used in the previous experiment, for each combination of evaluation function and heuristic function, applying a cutoff time of 6,000 milliseconds.

[2]https://github.com/drwiner/gdpop

Results The plan selection functions that perform best on the depth request are $\mathcal{E}^1$, $\mathcal{E}^2$, and $\mathcal{E}^3$ with the add-reuse heuristic. $\mathcal{E}^3$ also performs well with the number of open conditions heuristic, but overall does not perform as consistently and does not solve all 8 problems. The zero heuristic ($h(\pi) = 0$) typically did not find solutions in time.

Conclusion

Hybrid planning techniques are popular in theory and practice, but (as also noted by Shivashankar et al. 2016) little effort has been devoted to guiding the search using hierarchical information in a planner-independent way. This planning paradigm is used in domains where hierarchical knowledge characterizes phenomena of interest that is not expressible in non-hierarchical formalisms; for example, the recognition of higher-level concepts on the basis of more primitive event information (Lesh, Rich, and Sidner 1999; Cardona-Rivera and Young 2017) or the generation of narratives where hierarchies represent communicative patterns and story plots (Winer and Young 2017). With our work, search in these domains can generate deep, low-cost solutions in a more principled manner.

One of the key obstacles we overcome with our approach is avoiding excessive search on the hierarchical depth attribute. A naive strategy is to simply discount composite tasks that add depth. However, this strategy causes the planner to always search through the space of plans that insert composite tasks before considering primitive tasks. The magnitude of the discount determines how excessively the planner searches in the direction of inserting composite tasks to repair open conditions. At some point, the discount is outweighed by the size of the plan and backtracking occurs such that shallower tasks are considered by the planner to make the same repairs. Although this approach prioritizes maximum depth, it is slow because it behaves like brute force search on the hierarchical depth attribute to find the best depth/cost ratio.

On the other hand, if the least-cost plans are always expanded first and no discount is offered for inserting tasks that add hierarchical depth (as in the standard case), then the planner will start shallow and insert composite tasks just when it is strictly more efficient. A *depth-balanced* approach neither over-discounts nor under-discounts hierarchical depth. In this work, we formulate a dynamic discount for depth that exploits the hierarchical depth of the sub-trees of composite tasks to guide the planner to "deep" solutions. When a plan is "shallow" (hierarchically), deep composite tasks are discounted, but as the plan becomes "deeper", this discount recedes. We introduced a new way to measure plan depth, and used this measurement as part of a search strategy for decompositional planning where deep solutions are preferred. Our preliminary evidence supports a claim that our dynamic discount is useful for achieving a depth-balanced approach. Our method may also be useful for state-space decompositional planning, which has historically suffered from similar issues.

Importantly, we tested our scoring functions on a single planning domain. To verify that our results are generalizable, we also need to test the scoring functions with different hierarchical domains. The effects of differently structured hierarchical knowledge is less clear than primitive-only domains (Chrpa, McCluskey, and Osborne 2015); an evaluation with such differently structured hierarchical knowledge warrants a follow-up investigation that is beyond our scope here. This study would compare domains where the amount of decomposition knowledge provided as input is controlled, and its effects on the search for deep solutions is examined.

References

Bechon, P.; Barbier, M.; Infantes, G.; Lesire, C.; and Vidal, V. 2014. HiPOP: Hierarchical Partial-Order Planning. In *Proceedings of the 7th European Starting AI Researcher Symposium at the 21st European Conference on Artificial Intelligence*, 51–60.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. In *Proceedings of the 7th Annual Symposium on Combinatorial Search*, 35–43.

Bordwell, D.; Thompson, K.; and Smith, J. 1997. *Film Art: An Introduction*. McGraw-Hill New York.

Cardona-Rivera, R. E., and Young, R. M. 2017. Toward combining domain theory and recipes in plan recognition. In *Proceedings of the Plan, Activity, and Intent Recognition Workshop at the 31st AAAI*, 796–803.

Christianson, D. B.; Anderson, S. E.; He, L.-w.; Salesin, D. H.; Weld, D. S.; and Cohen, M. F. 1996. Declarative camera control for automatic cinematography. In *AAAI/IAAI, Vol. 1*, 148–155.

Christie, M.; Olivier, P.; and Normand, J.-M. 2008. Camera control in computer graphics. *Computer Graphics Forum* 27(8):2197–2218.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2015. On the Completeness of Replacing Primitive Actions with Macro-actions and its Generalization to Planning Operators and Macro-operators. *AI Communications* 29(1):163–183.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving Hierarchical Planning Performance by the Use of Landmarks. In *Proceedings of the 26th AAAI*, 1763–1769.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and Expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1123–1128.

Garoufi, K. 2014. Planning-based models of natural language generation. *Language and Linguistics Compass* 8(1):1–10.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd IJCAI*, 1955–1961.

He, L.-w.; Cohen, M. F.; and Salesin, D. H. 1996. The virtual cinematographer: a paradigm for automatic real-time camera control and directing. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 217–224. ACM.

Jhala, A., and Young, R. M. 2010. Cinematic visual discourse: Representation, generation, and evaluation. *IEEE Txn on Comp. Intelligence and AI in Games* 2(2):69–81.

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid Planning for Partially Hierarchical Domains. In *Proceedings of the 15th National Conference on Artificial Intelligence*, 882–888.

Lee-Urban, S. M. 2012. *Hierarchical Planning Knowledge for Refining Partial-Order Plans*. Ph.D. Dissertation, Lehigh University.

Lesh, N.; Rich, C.; and Sidner, C. L. 1999. Using plan recognition in human-computer collaboration. In *Proceedings of the 7th International Conference on User Modeling*, 22–32.

Prince, G. 2003. *A Dictionary of Narratology*. University of Nebraska Press.

Roth, E. M.; Hanson, M. L.; Hopkins, C.; Mancuso, V.; and Zacharias, G. L. 2004. Human in the loop evaluation of a mixed-initiative system for planning and control of multiple uav teams. In *Proceedings of the Human Factors and Ergonomics Society 48th Annual Meeting*, 280–284.

Schirner, G.; Erdogmus, D.; Chowdhury, K.; and Padir, T. 2013. The future of human-in-the-loop cyber-physical systems. *Computer* 46(1):36–45.

Shivashankar, V.; Alford, R.; Roberts, M.; and Aha, D. W. 2016. Cost-optimal algorithms for planning with procedural control knowledge. In *Proceedings of the 22nd European Conference on Artificial Intelligence*, 1702–1703.

Sperber, D., and Wilson, D. 1987. Précis of relevance: Communication and cognition. *Behavioral and brain sciences* 10(4):697–710.

To, S. T.; Langley, P.; and Choi, D. 2015. A Unified Framework for Knowledge-Lean and Knowledge-Rich Planning. In *Proceedings of the 3rd Annual Conference on Advances in Cognitive Systems*.

Weld, D. 1994. An Introduction to Least Commitment Planning. *AI Magazine* 15(4):27.

Winer, D. R., and Young, R. M. 2017. Merits of Hierachical Story and Discourse Planning with Merged Languages. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Younes, H. L., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.

Young, R. M.; Ware, S.; Cassell, B.; and Robertson, J. 2013. Plans and planning in narrative generation: a review of plan-based approaches to the generation of story, discourse and interactivity in narratives. *Sprache und Datenverarbeitung, Special Issue on Formal and Computational Models of Narrative* 37(1-2):41–64.

Young, R. M.; Pollack, M. E.; and Moore, J. D. 1994. Decomposition and causality in partial-order planning. In *Proceedings of the Conference on Artificial Intelligence Planning Systems*, 188–194.

Assumption-based Decentralized HTN Planning

Ugur Kuter and **Robert P. Goldman** and **Josh Hamell**
SIFT, LLC
319 1st Ave N., Suite 400,
Minneapolis, MN 55401, USA
{ukuter, rpgoldman, jhamell}@sift.net

Abstract

This paper describes our approach to decentralized planning via Hierarchical Task Networks (HTNs), which we call Autonomy and Rationale Coordination Architecture for Decentralized Environments (ARCADE). ARCADE is a decentralized AI planning framework that can incorporate a number of SHOP2 HTN planner instances. Each SHOP2 instance may have a different HTN planning domain definition than the others in the framework. ARCADE does not assume full communications among the planners. For this reason, ARCADE planners must make and manage assumptions about parts of the world state that are not visible to them, including the tasks and plans of other planners. The individual planners also must operate asynchronously, and may receive new tasks, either from outside, or from other planners in ARCADE.

In this paper, we describe our assumption-based planning approach and how ARCADE coordinates multiple, asynchronously interacting HTN planners, using *assumptions* and *task queues*. We first present a formal framework, Assumption-based, Decentralized Total-order Simple Task Network (DTSTN) planning, based on Total-order Simple Task Network planning. This is necessary because of our use of SHOP2-style task semantics, instead of goal semantics. Then we describe the ARCADE framework, and how it implements the framework. Finally, we present preliminary experimental results in a simplified air operations planning domain, which shows that ARCADE realizes the expected speed-up when applied to weakly coupled planning problems. We conclude with directions for future work.

Introduction

Existing distributed and multi-agent planning systems (Torreño et al. 2017) typically focus on deterministic planning problems, with relatively simple models. They also typically assume a *single* overall planning task that must be distributed among multiple agents. Most practical applications for decentralized planning (e.g., military operations, UAV planning, and others) involve independent planners and reasoners that are responsible for accomplishing different tasks under large-scale uncertainty, while communicating their intentions and coordinating their actions. These planners often are not handed a single, large problem to be decomposed and then solved. Instead, these planners often receive their own planning problems to solve based on the organizational structures in which they are embedded (e.g., logistics and manufacturing systems separately plan to secure inputs and to make products). They may also receive additional tasks at runtime.

The problems we are interested in also involve limited and unreliable communications. Thus, our planners must operate under assumptions about peer decisions and states, where knowledge is not fully shared. Finally, in these applications the classical assumption of complete information and predictability is typically difficult or impossible to apply. For example, Seuken and Zilberstein (2008) address partial-observability and uncertainty during planning; however, these approaches cannot scale up to the large-scale planning problems and use closed-world formalisms.

Autonomy and Rationale Coordination Architecture for Decentralized Environments (ARCADE) is a decentralized planning architecture that allows multiple SHOP2 (Goldman and Kuter 2018a; Nau et al. 2003) HTN planner instances to generate plans for planning tasks concurrently and asynchronously. Each planner may generate plans for tasks issued by other planners or received as input from outside. Our contributions in this paper are as follows:

- We present a formalism for assumption-based HTN planning, which allows SHOP2 to generate plans for execution in the presence of other (cooperative) agents, when agent-to-agent communications are unreliable and the environment is not fully observable.
- We describe ARCADE, our decentralized planning framework based on the above formalism. ARCADE takes as input a number of HTN planning problem specifications for all or a subset of the planners. ARCADE then coordinates the asynchronous operations of multiple SHOP2 instances.
- We describe how ARCADE communicates tasks to the planners by generating new HTN planning problem specifications, and by publishing those specifications. The planners can sign up to meet those requests if the tasks involved belong to the domain descriptions of those planners, and involve domain entities which those planners control. During the decentralized planning process, ARCADE ensures the plans generated in this way are sound and consistent, brokering solutions to conflicts between the decisions made by different planners.
- We are currently using ARCADE in various Air Operations planning scenarios. We present our preliminary experi-

ments and results in a high-level version of this domain, developed for publication purposes. Our preliminary results are promising: decentralized planning shows substantial scalability improvements over centralized planning via Shop2, as one would expect. We also provide a representative result on the experiments we are conducting on the various individual components of Arcade.

In the immediately following section we start by reviewing totally ordered simple task network (TSTN) planning. We then build our framework, *Assumption-based, Decentralized* TSTN planning, on the basic TSTN definitions. Critically, these definitions allow us to characterize what it means for Decentralized TSTN (DTSTN) plans to be consistent with each other. We then explain the Arcade approach, which builds consistent sets of DTSTN plans, using assumptions to lazily bind resources to tasks, and to enable agents to reason in the context of beliefs about each others' likely actions and state. We present preliminary experimental results that show the efficiency of our approach. Finally, we conclude with a review of related work and conclusions (including future directions).

Preliminaries: TSTN planning

We use the same definitions for logical substitutions, atoms, constant and variable symbols, positive and negative literals in a finite function-free first-order language, as in (Ghallab, Nau, and Traverso 2004). A *state* is a collection, s, of ground atoms. In our work, we adopt a restricted case of HTN planning called *Total-order Simple Task Network* (TSTN) planning (Ghallab, Nau, and Traverso 2004):[1]

```
(:action start-order
 :parameters
  (?o - order ?n ?n1 - count)
 :precondition (and
                 (waiting ?o)
                 (stacks-avail ?n)
                 (next-count ?n1 ?n))
 :effect (and (not (waiting ?o))
              (started ?o)
              (not (stacks-avail ?n))
              (stacks-avail ?n1)))
```

Figure 1: An example operator schema from the openstacks domain (Helmert, Do, and Refanidis 2010). Shop2 can use PDDL action definitions.

- A TSTN domain is a quadruple:
$$\mathcal{D} = \langle \mathrm{ops}(\mathcal{D}), \mathrm{tasks}(\mathcal{D}), \mathrm{meths}(\mathcal{D}), \omega(\mathcal{D}) \rangle$$
- Each operator $o \in \mathrm{ops}(\mathcal{D})$ is a triple
$$o = \langle \mathrm{name}(o), \mathrm{precond}(o), \mathrm{effects}(o) \rangle$$
where name(o) is a *task* (see below), and precond(o) and effects(o) are sets of literals called o's *preconditions* and *effects*. See Figure 1 for an example operator schema, and Figure 2 for an operator. If a state s satisfies precond(o), then o is *executable* in s, producing the state $\gamma(s, o) = (s - \{\text{all negated atoms in effects}(o)\}) \cup \{\text{all non-negated atoms in effects}(o)\}$.

```
(:action (start-order o1 s2 s1)
 :precondition (and
                 (waiting o1)
                 (stacks-avail s2)
                 (next-count s2 s1))
 :effect (and (not (waiting o1))
              (started o1)
              (not (stacks-avail s2))
              (stacks-avail s1)))
```

Figure 2: Operator from the openstacks schema in Figure 1. In this case `o1`, `s1`, and `s2` $\in \omega(\mathcal{D})$, and the task is `(start-order o1 s2 s1)`.

- $\mathrm{tasks}(\mathcal{D})$ is the finite set of ground tasks, such that $\mathrm{tasks}(\mathcal{D}) = \mathrm{prims}(\mathcal{D}) \cup \mathrm{comps}(\mathcal{D})$, where $\mathrm{prims}(\mathcal{D})$ is the set of primitive tasks and $\mathrm{comps}(\mathcal{D})$ is the (disjoint) set of nonprimitive (or complex) tasks in the planning domain.

  A task, t, is a symbolic representation of an activity. Syntactically, a task looks like a term (functor and arguments from the universe of the domain). If t is also the name of an operator, then τ is *primitive*; otherwise τ is *nonprimitive*. Primitive tasks can be instantiated into actions, and nonprimitive tasks need to be decomposed into subtasks.
- $\omega(\mathcal{D})$ is the universe of entities in the planning domain. We have seen in Figures 1 and 2, that $\omega(\mathcal{D})$ is used to form tasks (and hence operator and method names).
- A *method*, m, is a prescription for how to decompose a task into subtasks. m is a tuple:
$$m = \langle \mathrm{task}(m), \mathrm{precond}(m), \mathrm{subtasks}(m) \rangle,$$
where task$(m) \in \mathrm{tasks}(\mathcal{D})$ is the task m can decompose, precond(m) is a set of preconditions, and subtasks$(m) = (t_1, \ldots, t_j), t_i \in \mathrm{tasks}(\mathcal{D})$ is a sequence of subtasks, the *expansion* of task(m). See Figure 3 for an example.

```
(:pddl-method (open-all-stacks)
  open-a-stack-and-recurse
  (exists (?n ?n1 - count)
    (and (stacks-avail ?n)
         (next-count ?n ?n1)))
  (:ordered (open-new-stack ?n ?n1)
            (open-all-stacks)))
```

Figure 3: Example method from the openstacks domain for the task `(open-all-stacks)`. The precondition is that there be a stack available (an `?n1` that is not yet open). The subtasks are to open a new stack, and then open any remaining stacks, recursively. This is a lifted method schema, corresponding to multiple ground methods.

A TSTN planning problem is a tuple: $P = \langle \mathcal{D}, s_0, T_0 \rangle$, where $\mathcal{D}$ is a TSTN planning domain, s_0 is the initial state,

[1] In future work, we plan to generalize this – see discussion in the Conclusions.

and T_0 is an initial sequence of tasks. (1) If T_0 is the empty sequence, ϵ then the only solution is the empty plan $\pi = \langle\rangle$, and π's *derivation* (the sequence of actions and method instances used to produce π) is $\delta = \langle\rangle$. If the current set of tasks is $t_1 \dots t_n$ and (2) t_1 is a primitive task, there is an operator α with $\text{name}(\alpha) = t_1$, and α is executable in s_i producing a state s_1, then if $\mathcal{P}' = \langle \mathcal{D}, s_1, T' \rangle$ has a solution π with derivation δ, then the plan $\alpha \bullet \pi$ is a solution to w_i (where $\bullet$ is concatenation) whose derivation is $\alpha \bullet \delta$. (3) If t_1 is nonprimitive and there is a method m such that $\text{task}(m) = t_1$, and if s_0 satisfies $\text{precond}(m)$, and if $P' = (s_0, \text{subtasks}(m) \bullet T', O, M)$ has a solution π with derivation δ such that δ only uses objects from the agent i's tasks T, then $\alpha \bullet \pi$ is a solution to P and its derivation is $m \bullet \delta$.

Assumption-based Decentralized TSTN Planning

In Arcade, an *assumption* is an ordered pair $e = \langle \mathsf{cond}, \mathsf{cost} \rangle$, where cond is a literal and cost is a non-negative real number that denotes the cost of validating the cond. More precisely, the cost is a heuristic estimate of the cost of checking to see whether the assumption is guaranteed to hold at run time or not. Two assumptions, $e_1 = \langle \mathsf{cond}_1, \mathsf{cost}_1 \rangle$ and $e_2 = \langle \mathsf{cond}_2, \mathsf{cost}_2 \rangle$ are *inconsistent*, if either (1) $\mathsf{cond}_1 = \text{not}(\mathsf{cond}_2)$, or *vice versa*; or (2) $\mathsf{cond}_1 = \mathsf{cond}_2$ and $\mathsf{cost}_1 \neq \mathsf{cost}_2$. Assumptions that are not inconsistent are consistent.

A *decentralized TSTN planning agent* is a tuple of the form $A = \langle \mathcal{D}, \omega(A), \text{ops}(A), \text{meths}(A) \rangle$, where $\mathcal{D}$ is a TSTN planning domain as defined above, $Q(A)$ is a *task agenda*, $\omega(A) \subseteq \omega(\mathcal{D})$ is the subset of entities that the agent can manipulate, $\text{ops}(A) \subseteq \text{ops}(\mathcal{D})$ and $\text{meths}(A) \subseteq \text{meths}(\mathcal{D})$ are the sets of tasks, operators, and methods for this agent. The task agenda of a planning agent is a basic queue data structure, which only allows tasks to be accomplished in chronological order.

Note that the tasks of an agent A, $\text{tasks}(A) \subseteq \text{tasks}(\mathcal{D})$ is defined as $\bigcup_{o \in \text{ops}(A)} \text{name}(o) \cup \bigcup_{m \in \text{meths}(A)} \text{task}(m)$. In other words, A can decompose a task $t \in \text{tasks}(A)$ as long as it has either an operator or a method definition for t. Otherwise, A pauses its planning process, and requests Arcade to find another agent that can perform t for it. When Arcade receives such a request, Arcade sends t to the other planning agents in the framework. If there exist one or more agents that can achieve t, these agents can start planning for t via their own methods and operators. For example, in the openstacks shipping domain, we could imagine an agent that is responsible for shipping the orders, and an agent that is responsible for adding the products to stacks.

We define an *assumption-based planning state*, s, as a collection of ground literals, $\text{facts}(s)$, and assumptions $\text{assumps}(s)$. The collection of literals in an assumption-based state describes the facts that a planning agent knows to be either true or false. The assumptions model the beliefs (as opposed to the knowledge) of the agent. That is, if an assumption of the form $\langle \mathsf{cond}, \mathsf{cost} \rangle$ is in the agent's state, where cond may be a positive or negative ground literal, this means that the agent makes a belief assertion, however, the agent does not know whether that assertion is correct or not. Validating that assertion is costly during planning; if the agent's assertion is proved to be wrong, any assumption-based plan that is conditional on assertion must be repaired. The cost value in the assumption estimates this cost.

An assumption-based planning state is *consistent* if: (1) the known facts, $\text{facts}(s)$ are consistent, (2) there does not exist an assumption $a_\perp \in \text{assumps}(s)$ such that $a_\perp = \langle \mathsf{cond}, \mathsf{cost} \rangle$ and the negation of cond is in $\text{facts}(s)$ (3) the set of assumptions is consistent.

An *assumption-based plan*, κ, is a sequence of pairs of the form $\langle b, \alpha \rangle$ where b is an assumption-based state and α is an action. Two assumption-based plans, κ_1 and κ_2, are consistent if (1) κ_1 and κ_2 are individually consistent, (2) κ_1 and κ_2 are well-formed TSTN plans, and (3) for all $(b_1, \alpha_1) \in \pi_1$ and $(b_2, \alpha_2) \in \pi_2$, b_1 and b_2 are consistent. We trivially generalize the definition of pairwise consistency to group consistency being the case of all pairs being consistent.

A *decentralized TSTN planning problem* is a tuple of the form $P = (\mathcal{A}, B_0, \mathcal{T}_0)$ where $\mathcal{A}$ is the finite set of planning agents for the decentralized planning problem. B_0 is a collection of *initial* assumption-based states such that for each $A \in \mathcal{A}$, there exists an assumption-based state $b_0(A) \in B_0$. Similarly, $\mathcal{T}_0$ is a collection of TSTNs such that there is a TSTN $T_0(A) \in \mathcal{T}_0$ for each agent $A \in \mathcal{A}$. A *solution* to a decentralized TSTN planning problem P is a collection of assumption-based plans that are consistent.

Decentralized Planning Framework

We designed our decentralized planning framework with the following objectives in mind:

1. *Asynchronous decentralization and planning:* Different Shop2 instances must be able to receive their TSTN planning problems at different points in time during decentralized planning and they must be able to work on those problems concurrently, and each at its own pace.
2. *Task-centric assumption-based coordination:* Shop2 instances must be able to exchange subtasks during planning, based on the assumptions each makes and whether or not a planner is capable of generating plans for specific tasks.
3. *Hierarchical localized plan adaptation and repair:* Each Shop2 instance in the decentralized planning framework must use localized replanning and plan repair algorithms (Goldman and Kuter 2018b) to provide consistency and correctness over its assumptions, which might be invalidated by the decisions and plans made by other Shop2 instances.

We will focus on objectives (1) and (2) in the rest of this paper. We discuss objective (3) in a forthcoming, related paper (Goldman and Kuter 2018b). Given a decentralized TSTN planning problem $P = (\mathcal{A}, B_0, \mathcal{T}_0)$ as defined above, a Shop2 planning agent. A_i, starts to generate solution plans to the tasks in its agenda $Q(A_i)$. Initially, this agenda contains only the initial task sequence T_0 that is specified for this agent in the input planning problem description. Given its initial state b_0, A_i extracts a task t_0 from its agenda, creates a local

TSTN planning problem $P_i = (\mathcal{D}, b_0, \{t_0\})$, and calls SHOP2 on this local TSTN planning problem to generate a solution plan.

During local TSTN planning, if A_i generates a task t such that $t \notin \text{tasks}(A_i)$,[2] then A_i escalates this task to ARCADE, which in turn publishes t for all of the other agents in $\mathcal{A}$. If there exists at least one other agent, say A_j, in ARCADE such that $t \in \text{tasks}(A_j)$, A_j will insert t into its own task agenda, $Q(A_j)$. If no agent is able to generate a plan for t, then ARCADE notifies A_i and A_i backtracks to consider other alternative task decompositions. This is done by imposing a timeout, a settable parameter, on agent-to-agent requests.

In our current system, if A_i escalates a task t to ARCADE for other agents to plan for t, A_i pauses its planning until a response comes back. This is done to maintain a consistent planning state throughout all the tasks in A_i's agenda while different planners work on different tasks at different times. A_i incorporates the supplying agent ($agent_j$)'s plan into its own plans. This involves progressing the current planning state of the A_i by applying the actions in the plan being incorporated.

If multiple agents generate plans for t, A_i selects one of them, and drops the others. A_i (through ARCADE), informs any un-selected agents that they can drop their plans for t. In our current implementation, the choice over the plans generated by the other agents is greedy: A_i selects the first plan it receives from the other agents and rejects any other responses. We are developing a theory for how to use the assumption costs to make such selections. ARCADE already uses the cost of validating an assumption as a way to de-clobber plans of different agents, if necessary (see below), and a similar mechanism can be used as a heuristic to choose between the plans of different agents.

In ARCADE, a planning agent **generates assumptions** in two cases: (1) while it is accomplishing a task and (2) the agent generates assumptions while incorporating another agent's plans into its solution. We discuss these two cases below.

Generating assumptions during planning Although our formal discussion describes TSTNs and DTSTSs in a propositional context, SHOP2 is, in fact, a *lifted* (first order) planner, and performs many of its tasks using *unification*.

During TSTN planning for a task t with precondition $p(x)$, for example, if SHOP2 cannot find a substitution for x satisfying $p(x)$, it normally backtracks. In ARCADE, we modified this behavior so that SHOP2 can generate an assumption, instead of backtracking, and continuing the search with that assumption asserted into its state representation.

For example, suppose t is a task for taking an image, for which it must assign a UAV with appropriate instruments (e.g., cameras). In the current state it fails to do so, because its planning state is incomplete, and it cannot determine whether the suitable UAV will be available. Standard SHOP2 would backtrack at this point. In ARCADE, SHOP2 has the alternative of generating an *assumption*, here for example assuming that uav1 will be available: $\langle \mathsf{available}(\mathsf{uav1},\ 1600), 1\rangle$.

One of the key challenges for a planner is to determine whether an assumption is too specific or too general. Most modern planning systems eagerly ground variables in their action schemas. IPC planners typically preprocess and ground problem specifications and domain models *a priori*. Lifted planners such as SHOP2 ground variables on the fly but they do so at the first point where a ground value is matched during search. If those variable-binding choices do not lead to solution plans later on in search, the planner backtracks and tries other possible groundings (Nau et al. 2003). This is a major scalability issue, even for SHOP2; as has been shown in experiments over a decade now, SHOP2's performance can degrade exponentially (Nau et al. 2003).

In decentralized planning, this performance degradation is more dire because of uncertainty and incomplete information induced due to the operations of multiple planners. In addition, backtracking over decisions that involve other agents is even more time-consuming than it is in centralized planning.

In particular, an agent A_i knows the set of constant symbols in $\omega(A_i)$, but it does not know about $\omega(A_j)$ for $j \neq i$. Thus, it cannot generate an assumption that involves grounded conditions about other agent's planning states. To address this challenge, we have developed a *late-binding* approach for SHOP2 to use logical *skolemization* as in automated reasoning works (Genesereth and Nilsson 1987). In particular, SHOP2 delays binding a variable symbol that appears in an assumption condition; instead, SHOP2 replaces it with a skolem function that specifies the properties, as constraints, of the constant that should be bound to that variable for a sound plan. After A_i's SHOP2 generates a plan with skolem functions in it as a solution, ARCADE post-processes the plan and generates variable bindings according to the generated constraints during planning. Our preliminary experiments in the subsequent section show the potential benefits of this approach. In principle, however, post-processing may still fail to generate bindings successfully for some of the skolem functions. In that case, ARCADE treats the binding failure as a plan-failure discrepancy and triggers its plan adaptation and repair process.

Coordinating over other agents' assumptions When a planning agent A_i receives an assumption-based plan κ_j from another agent A_j and attempts to incorporate κ_j into its plans, it may find that κ_j is inconsistent with its own plan, κ_i. When merging, each agent will verify the consistency of the plans, using the definitions given above. If an inconsistency is found between two assumptions, ARCADE compares the cost of validating the inconsistent assumptions. If A_i's own cost of adapting to assumption violation is greater than those of A_j, then ARCADE notifies A_j about the contradictions and the cost models over them, requiring A_j to adapt to its assumptions. Otherwise, A_i casts the contradiction as a plan discrepancy (using our plan repair framework (Goldman and Kuter 2018b)) and adapts its own plans. In ARCADE, currently all ties are broken randomly.

If $agent_j$ cannot adapt its plans to alleviate the contrac-

[2]This will happen when A_i expands a task t' using a method with t as a subtask.

tion, then it returns a failure. At this point, A_i attempts to adapt its plan even if it was deemed heuristically more costly. If A_i generates a solution, then it incorporates it into its assumption-based plans. Otherwise, the goal is not satisfied and A_i returns failure.

Calculating costs of assumption violations When an agent generates assumptions, it also rapidly calculates a cost estimate. This estimate attempts to characterize how much work it would be to adapt its plans (if it can at all), if the assumption is violated. In ARCADE, this is done by a combination of plan critiquing and generating adaptation options (i.e., alternative plan repairs that are different from each other in terms of the objects used and how the adapted tasks are accomplished). The latter is the topic of a forthcoming paper. We have described our work on plan critiquing in (Goldman, Kuter, and Schneider 2012; Mueller et al. 2017) in detail; we summarize it briefly below.

Our plan critiquing system, Murphy (Goldman, Kuter, and Schneider 2012), generates counterexamples for a SHOP2 plan to explain possible breakdown cases for that plan. In ARCADE, Murphy critiques plans to assess assumption violations due to the plans and assumptions made by the other planners in the framework. Murphy translates a plan into a "counter-planning" problem, combining a representation of the initial plan with the definition of a set of uncontrolled actions. These uncontrolled actions may be the actions of other agents in the ARCADE framework, actions of the exogenous agents in the environment, either friendly, indifferent or hostile, or they may be exogenous events that simply occur. The result of this translation is a disjunctive planning problem that we further process in order to play into the strengths of existing classical planners. Using this formulation, a classical planner can find counterexamples that illustrate ways a plan may go awry.

In ARCADE, the translation yields a new counter-planning problem and domain, in which goals are logically-negated assumption conditions. For example, an assumption of the form (cond cost) produces a goal literal of the form ($\neg$cond). Such counter-planning goals can then be solved to find a counterexample or, if searched exhaustively without finding a counterexample, indicates that no counterexample exists. There are three components to the translation process: (1) generating a "skeleton" for the original plan that ensures that the original plan actions occur in the proper order; (2) formulating a goal expression that will cause a planner to search for a counterexample and (3) encoding the resulting disjunctive goal expression in a way that is amenable to the use of a standard PDDL planner.

If a counterexample is found, the agent generates multiple adaptations of the plan, treating the counterexample as a discrepancy that breaks the plan. For each of these adaptations, the agent calculates the cost of the new plans. The adaptation with the maximum cost is used to update the cost of violating the assumption. Intuitively, this approach calculates the worst case that i's plans must be adapted if they are violated by other agent's assumptions and plans. As described above, the cost estimate of violating the assumption is then used to compare assumptions of different ARCADE agents and guide coordination and deconfliction of the agents' knowledge states.

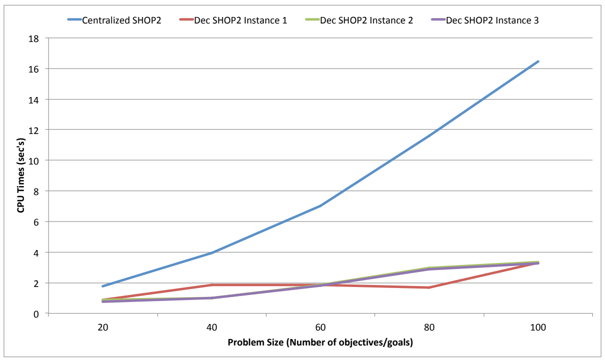

Figure 4: Comparison of run times using SHOP2 in a centralized fashion versus using our decentralized approach.

Implementation and Evaluations

We have implemented our ARCADE framework in Common Lisp, using SHOP2 as the planner for the planning agents. ARCADE allows an arbitrary number of planning agents; our current experiments use a 4-planner instance of this framework. We use an air-operations planning, command, and control domain, that we have developed, and which is a generalization of the OpenStacks domain from the International Planning Competition (Helmert, Do, and Refanidis 2010). In this domain, planning operators specify air missions to achieve a given set objectives (i.e., locations), their scheduling, and resource usage to create plans that can be given the human military operators to execute. The planning problems in this domain include a varying number of aircraft, bases, objectives, locations on a physical map layout. Our HTN methods encode strategies that describe how generate groups of missions to achieve all of the objectives under different conditions of the world.

Using the above framework, we are currently conducting several experiments with ARCADE to evaluate the approach's performance. Below, we present and discuss some representative results of our experiments.

Figure 4 shows our preliminary results to confirm the runtime scalability benefits of a decentralized approach, in contrast to a single-agent, centralized planning. Here, we use the same setting as described above, planning problems that involved tens of aircraft distributed across five bases. The x-axis shows the number of objectives we varied for evaluation purposes.

This experiment was intended as a sanity check on our decentralized planning approach. Decentralized planning over non-conflicting tasks should achieve near linear (optimal) speed-up over centralized planning. As shown in Figure 4, our experiments confirmed this expectation: as a function of increasing problem size (i.e., increasing number of tasks to be accomplished), the runtime performance of planner nodes in the decentralized approach remains linear, whereas

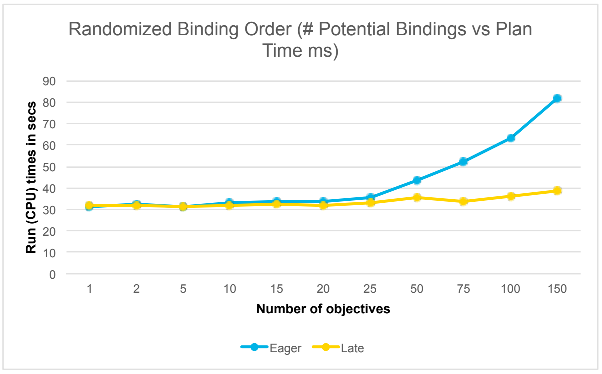

Figure 5: Comparison of eager (original) variable binding strategy in SHOP2's backtracking search and late binding.

Shop2's runtimes increase substantially.

In addition to the scalability of the overall framework, we are also experimenting to identify which factors are key drivers of the performance of individual Shop2 instances in our air-operations problems. One of the factors we explored is the impact of late-binding in the assumptions generated during planning. Figure 5 shows the results of a preliminary experiment using the same setting as above. The x-axis again denotes the number of objectives we have varied for the purposes here. This experiment compares the original eager-binding approach of Shop2 with a late-binding strategy in which we randomly varied the variables that Shop2 eagerly binds and those variables it skolemizes. The results suggest a significant performance improvement with the latter approach.

Related Work

Our work builds on the great strides on distributed, decentralized and multi-agent planning works over the years. Torreño, et al. (2017) provide an excellent up-to-date survey covering historical developments and the current state of the art. Unlike most distributed, multi-agent planning work, which focuses on domain-independent deterministic planning and heuristic search, we focus on HTN planning (Ghallab, Nau, and Traverso 2004; Nau et al. 2003; Tate, Drabble, and Kirby 1994; Wilkins 1988). HTN planning is critical to our applications, because it permits our system to capture doctrinal and procedural tasks, constraints and communication strategies. Modern HTN planners also incorporate considerably greater expressive power than standard IPC/PDDL-style planners. They typically include: capabilities for calling attached procedures, making axiomatic inferences, and performing numeric computations, all of which are critical for this approach.

Previous work on hierarchical planning and scheduling includes priority-based task planning for UAVs (Musliner et al. 2011), hybrid HTN planning and scheduling (e.g., (Schattenberg 2009; Elkawkagy et al. 2010; Bercher et al. 2014; Bercher, Keen, and Biundo 2014), SharedPlans (Grosz and Kraus 1996), multi-agent HTN planning (Dix et al. 2003; Gancet et al. 2005; Elkawkagy and Biundo 2011), HTNs with temporal reasoning (Goldman 2006; Fdez-Olivares et al. 2006), and HTNs for planning under uncertainty in world states (e.g., (Kuter et al. 2009)). Although these systems have proven themselves on research benchmarks, they (1) make specific, simplifying assumptions, which are also independent from each other and do not align well, (2) do not scale up to our planning problems in the air operations planning domain, and (3) cannot perform resilient synchronized planning.

Our work is most similar to the Shared Activity Coordination (SHAC) system of Clement and Barrett (2003). SHAC was built to support continual planning and execution for NASA missions with multiple different assets, multiple stakeholders, and limited communications. However, the nature of the communications limitations in SHAC is quite different from that of Arcade: theirs comes from the physics of the domain (orbits, rotations, etc.), and is largely predictable, whereas ours comes from unpredictable external factors. Also, their communications issues are generally planner to executor (probe), rather than planner to planner, like Arcade's. This shows in the differing nature of their consensus approach, and in their planning agents operating in regular cycles, unlike ours, and the SHAC agents do not require assumption-based planning.

Existing work on Multi-Agent STRIPS (MA-STRIPS) and its successors (Brafman and Domshlak 2008; Nissim, Brafman, and Domshlak 2010; Nissim and Brafman 2013) are directly related to Arcade. MA-STRIPS's concept of private/public actions to define an interface among the cooperating agents is similar to the assignment of subtasks with possible agents in our decentralized HTN models. One difference with formalism is that Arcade's agents do not coordinate to generate a single solution plan and interact to satisfy its causal model during planning, as MA-STRIPS and its successors are naturally designed to do as a classical planning approach. Instead, Arcade generates problem decomposition and refinement via TSTN planning and each agent generates plans for tasks. Another key difference between the MA-STRIPS family and Arcade is the concept of commitments, where the former aims to use commitments to ensure the correctness of the output multi-agent plan. Instead, Arcade treats most of the plan commitments of an agent as assumptions and ensures correctness via plan repair if those assumptions are broken by other agents' plans.

Note that an approach such as MA-STRIPS may be superior if the task involves taking a large planning problem and decomposing it into multiple subproblems for computational or even execution efficiency. These approaches would provide at least some support for automatic domain decomposition. In Arcade, on the other hand, the domain decomposition is assumed top be given. For a new situation, built from scratch, that would involve additional work. Arcade, however, was built for applications in large human organizations, where the structure of the organization and the scope of responsibility and authority are a given and a part of the problem to which the system must be adapted. This is why Arcade starts from a given problem decomposition – and indeed will adapt to the "plug and plan" addition of new agents.

Unlike other existing work that only focuses on planning in decentralized systems, our decentralized planning architecture ARCADE incorporates plan generation via HTNs with plan adaptation and critiquing. Closest to our approach is probably the Continuous Planning and Execution Framework (CPEF) (Myers 1999), which responds robustly to arbitrary changes in the world, by combining plan generation, execution monitoring, and repair capabilities. CPEF executes and monitors its plans using PRS (Georgeff and Ingrand 1989), and uses HTN planning from SIPE-2 (Wilkins 1988).

There are conceptual similarities between ARCADE and previous work on planning in partially-observable domains (Kaelbling, Littman, and Cassandra 1998; Bertoli et al. 2001; 2006; Kuter et al. 2007; Bonet and Geffner 2011). Both approaches deal with high volume of uncertainty in the planning state during planing time. The latter models the uncertainty explicitly, by leaving it in the solution policies when it is more expensive to resolve it. Our approach takes a chance on possibly unexpected outcomes and state conditions by making assumptions during planning time and appreciating the fact that those assumptions that the plans are conditioned upon can be violated during execution. To reduce uncertainty, ARCADE then uses both (1) planning-time plan critiquing capabilities (Goldman, Kuter, and Schneider 2012; Mueller et al. 2017) to foresee and avoid such failures and (2) rapid HTN plan repair algorithms (Goldman and Kuter 2018b).

Conclusions

We have described our ongoing work on decentralized planning and coordination. The basis of our approach is HTN planning domain definitions employed by multiple HTN planners, i.e., in this case SHOP2, that may be assigned to different tasks or may work on the same tasks in parallel. We are currently finalizing our formalism and conducting more experiments. We plan to investigate the performance and stability of ARCADE under varying conditions of decentralization and disturbance. We will include sensitivity analyses, varying the degree of decentralization of our problems, and assessing the capability (problems successfully solved) and the stability of our assumption-based planning approach. We will also vary the rate of perturbations (external changes to the world state, addition and deletion of new tasks) to assess the stability/volatility of our assumption-based planning approach. We will explore ways to conduct comparisons with existing multi-agent planning systems, MA-STRIPS in particular.

We would like to extend ARCADE to cover task networks that are not totally-ordered. Although planning algorithms and formalisms that can model partially-ordered HTNs exist (e.g., UMCP (Erol, Hendler, and Nau 1994), SHOP2 (Nau et al. 2003), PANDA (Bercher et al. 2017), and FAPE (Dvorak et al. 2014)), we have limited ourselves to TSTN planning as the basis of our formalism to simplify the criteria for correctness for the assumption-based plans.

Another future research direction is to incorporate temporal reasoning in assumption management and coordination. There are two aspects we plan to study: (1) the lifetime of the assumptions themselves: e.g., “if an agent does not hear back about an assumed condition in t time units since the assumption was made, it will cease to accept the assumed truth value”; and (2) temporal bounds on the period over which the assumptions are supposed to hold: e.g., A_i assumes that A_j is going to perform a particular action sometime between the time points t_1 and t_2 ($t_2 > t_1$); A_i must regard this assumption as violated if it is not confirmed by t_2.”

We will investigate both directions by borrowing the concept of information volatility from our previous work on Semantic Web Service Composition planning (Au, Kuter, and Nau 2005; Kuter et al. 2005). In this approach, temporal assumptions will model temporal uncertainty on a SHOP2 instance’s assumptions made over the tasks and plans over other planner instances in the framework. Another possible approach, perhaps complementing the first one, is to probabilistically assess the belief that a SHOP2 instance has in the assumptions regarding another planner will fulfill its commitments to its tasks and plans.

Acknowledgments. The work reported in this paper project is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8750-16-C-0182 for the Distributed Operations program. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not reflect the views of the AFRL. Cleared for public release, no restrictions.

Thanks to the anonymous referees for many helpful suggestions that has substantially improved our original manuscript.

References

Au, T.-C.; Kuter, U.; and Nau, D. S. 2005. Web service composition with volatile information. In *ISWC*.

Bercher, P.; Biundo, S.; Geier, T.; Hoernle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, repair, execute, explain-how planning helps to assemble your home theater. In *ICAPS*.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In Sierra, C., ed., *IJCAI*.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Seventh Annual Symposium on Combinatorial Search*.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI*.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2006. Strong Planning under Partial Observability. *Artificial Intelligence* 170:337–384.

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In *IJCAI*.

Brafman, R. I., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*.

Clement, B. J., and Barrett, A. C. 2003. Continual coordination through shared activities. In *AAMAS*. ACM Press.

Dix, J.; Muñoz-Avila, H.; Nau, D. S.; and Zhang, L. 2003. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence* 37(4):381–407.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Plan-Space Hierarchical Planning with the Action Notation Modeling Language. In *IEEE ICTAI*.

Elkawkagy, M., and Biundo, S. 2011. Hybrid multi-agent planning. In *German Conference on Multiagent System Technologies*, 16–28. Springer.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2010. Exploiting landmarks for hybrid planning. In *25th PuK Workshop Planen, Scheduling und Konfigurieren, Entwerfen*.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI*.

Fdez-Olivares, J.; Castillo, L.; Garcia-Perez, O.; and Palao, F. 2006. Bringing users and planning technology together, experiences in SIADEX. In *ICAPS*.

Gancet, J.; Hattenberger, G.; Alami, R.; and Lacroix, S. 2005. Task planning and control for a multi-uav system: architecture and algorithms. In *IEEE IROS*.

Genesereth, M. R., and Nilsson, N. J. 1987. *Logical foundations of Artificial Intelligence*. Springer.

Georgeff, M., and Ingrand, F. 1989. Decision-making in an embedded reasoning system. In *IJCAI*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Goldman, R. P., and Kuter, U. 2018a. Explicit stack search in SHOP2. Technical Report 2018-1, SIFT, LLC, Minneapolis, MN, USA.

Goldman, R. P., and Kuter, U. 2018b. Minimal perturbation plan repair for state-space HTN planning. Technical Report 2018-2, SIFT, LLC, Minneapolis, MN, USA.

Goldman, R. P.; Kuter, U.; and Schneider, A. 2012. Using classical planners for plan verification and counterexample generation. In *AAAI Workshop on Problem Solving Using Classical Planning*.

Goldman, R. P. 2006. Durative planning in HTNs. In *ICAPS*.

Grosz, B. J., and Kraus, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence* 86(2):269–357.

Helmert, M.; Do, M.; and Refanidis, I. 2010. Webpage for IPC-08. Retrieved most recently May 2018.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101(1-2):99–134.

Kuter, U.; Sirin, E.; Parsia, B.; Nau, D.; and Hendler, J. 2005. Information gathering during planning for web service composition. *Journal of Web Semantics*.

Kuter, U.; Nau, D. S.; Reisner, E.; and Goldman, R. 2007. Conditionalization: Adapting forward-chaining planners to partially observable environments. In *ICAPS 07 Workshop on Planning and Execution for Real-World Systems*.

Kuter, U.; Nau, D.; Pistore, M.; and Traverso, P. 2009. Task Decomposition on Abstract States for Planning under Nondeterminism. *Artificial Intelligence* 173:669–675.

Mueller, J. B.; Miller, C. A.; Kuter, U.; Rye, J.; and Hamell, J. 2017. A human-system interface with contingency planning for collaborative operations of unmanned aerial vehicles. In *AIAA Information Systems-AIAA Infotech@ Aerospace (2017-1296)*. AIAA Press.

Musliner, D.; Goldman, R. P.; Hamell, J.; and Miller, C. 2011. Priority-based playbook tasking for unmanned system teams. In *AIAA*. American Institute of Aeronautics and Astronautics.

Myers, K. L. 1999. A continuous planning and execution framework. *AI Magazine* 63–69.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Nissim, R., and Brafman, R. I. 2013. Cost-optimal planning by self-interested agents. In *AAAI*.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *AAMAS*.

Schattenberg, B. 2009. *Hybrid Planning And Scheduling*. Ph.D. Dissertation, Ulm University, Institute of Artificial Intelligence. URN: urn:nbn:de:bsz:289-vts-68953.

Seuken, S., and Zilberstein, S. 2008. Formal models and algorithms for decentralized decision making under uncertainty. In *AAMAS*.

Tate, A.; Drabble, B.; and Kirby, R. 1994. *O-Plan2: An Architecture for Command, Planning and Control*. Morgan-Kaufmann.

Torreño, A.; Onaindia, E.; Komenda, A.; and Štolba, M. 2017. Cooperative multi-agent planning: A survey. *ACM Computing Surveys (CSUR)* 50(6):84.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.

HEART: HiErarchical Abstraction for Real-Time Partial Order Causal Link Planning [*]

Antoine Gréa and **Laetitia Matignon** and **Samir Aknine**

Abstract

In recent years the ubiquity of artificial intelligence raised concerns among the uninitiated. The misunderstanding is further increased since most advances do not have explainable results. For automated planning, the research often targets speed, quality, or expressivity. Most existing solutions focus on one criteria while not addressing the others. However, human-related applications require a complex combination of all those criteria at different levels. We present a new method to compromise on these aspects while staying explainable. We aim to leave the range of potential applications as wide as possible but our main targets are human intent recognition and assistive robotics. The HEART planner is a real-time decompositional planner based on a hierarchical version of Partial Order Causal Link (POCL). It cyclically explores the plan space while making sure that intermediary high level plans are valid and will return them as approximate solutions when interrupted. These plans are proven to be a guarantee of solvability. This paper aims to evaluate that process and its results compared to classical approaches in terms of efficiency and quality.

Introduction

Since the early days of automated planning, a wide variety of approaches have been considered to solve diverse types of problems. They all range in expressivity, speed, and reliability but often aim to excel in one of these domains. This leads to a polarization of the solutions toward more specialized methods to tackle each problem. All of these approaches have been compared and discussed extensively in the books of Ghallab et al. (2004; 2016).

Partially ordered approaches are popular for their least commitment aspect, flexibility and ability to modify plans to use refinement operations (Weld, 1994). These approaches are often used in applications in robotics and multi-agent planning (Lemai and Ingrand, 2004; Dvorak et al., 2014). One of the most flexible partially ordered approaches is called *Partial Order Causal Link planning (POCL)* (Young and Moore, 1994). It works by refining partial plans consisting of steps and causal links into a solution by solving all flaws compromising the validity of the plan.

Another approach is *Hierarchical Task Networks (HTN)* (Sacerdoti, 1974) that is meant to tackle the problem using composite actions in order to define hierarchical tasks within the plan. Hierarchical domains are often considered easier to conceive and maintain by experts mainly because they seem closer to the way we think about these problems (Sacerdoti, 1975).

In our work, we aim combining HTN planning and POCL planning in such a manner as to generate intermediary high level plans during the planning process. Combining these two approaches is not new (Young and Moore, 1994; Kambhampati et al., 1998; Biundo and Schattenberg, 2001). Our work is based on *Hierarchical Partial Order Planning (HiPOP)* by Bechon et al. (2014). The idea is to expand the classical POCL algorithm with new flaws in order to make it compatible with HTN problems and allowing the production of abstract plans. To do so, we present an upgraded planning framework that aims to simplify and factorize all notions to their minimal forms. We also propose some domain compilation techniques to reduce the work of the expert conceiving the domain.

In all these works, only the final solution to the input problem is considered. That is a good approach to classical planning except when no solutions can be found (or when none exists). Our work focuses on the case when the solution could not be found in time or when high level explanations are preferable to the complete implementation detail of the plan. This is done by focusing the planning effort toward finding intermediary abstract plans along the path to the complete solution.

In the rest of the paper, we detail how the HiErarchical Abstraction for Real-Time (HEART) planner creates abstract intermediary plans that can be used for various applications. First, we discuss the motivations and related works to detail the choices behind our design process. Then we present the way we modeled an updated planning framework fitting our needs and then we explain our method and prove its properties to finally discuss the experimental results.

Motivations and Potential Applications

Several reasons can cause a problem to be unsolvable. The most obvious case is that no solution exists that meets the requirements of the problem. This has already been addressed by Göbelbecker

[*] Univ Lyon, Université Lyon 1, CNRS, LIRIS, UMR5205, F-69621, LYON, France (first.lastname@liris.cnrs.fr)

et al. (2010) where "excuses" are being investigated as potential explanations for when a problem has no solution.

Our approach deals with the cases of when the problem is too difficult to solve within tight time constraints. For example, in robotics, systems often need to be run within refresh rates of several Hertz giving the process only fractions of a second to give an updated result. Since planning is at least EXPSPACE-hard for HTN using complex representation (Erol et al., 1994), computing only the first plan level of a hierarchical domain is much easier in relation to the complete problem.

While abstract plans are not complete solutions, they still display a useful set of properties for various applications. The most immediate application is for explainable planning (Fox et al., 2017; Seegebarth et al., 2012). Indeed a high-level plan is more concise and does not contain unnecessary implementation details that would confuse a non-expert.

Another potential application for such plans is relative to domains that work with approximative data. Our main example here is intent recognition which is the original motivation for this work. Planners are not meant to solve intent recognition problems. However, several works extended what is called in psychology the *theory of mind*. That theory is the equivalent of asking "*what would **I** do if I was them ?*" when observing the behavior of other agents. This leads to new ways to use *inverted planning* as an inference tool. One of the first to propose that idea was Baker et al. (2007) that use Bayesian planning to infer intentions. Ramirez and Geffner (2009) found an elegant way to transform a plan recognition problem into classical planning. This is done simply by encoding temporal constraints in the planning domain in a similar way as Baioletti et al. (1998) describe it to match the observed action sequence. A cost comparison will then give a probability of the goal to be pursued given the observations. Chen et al. (2013) extended this with multi-goal recognition. A new method, proposed by Sohrabi et al. (2016), makes the recognition fluent centric. It assigns costs to missing or noisy observed fluents, which allows finer details and less preprocessing work than action-based recognition. This method also uses a meta-goal that combines each possible goal and is realized when at least one of these goals is satisfied. Sohrabi *et al.* state that the quality of the recognition is directly linked to the quality and domain coverage of the generated plans. Thus guided diverse planning[1] was preferred along with the ability to infer several probable goals at once.

Related Works

HTN is often combined with classical approaches since it allows for a more natural expression of domains making expert knowledge easier to encode. These kinds of planners are named **decompositional planners** when no initial plan is provided (Fox, 1997). Most of the time the integration of HTN simply consists in calling another algorithm when introducing a composite operator during the planning process. In the case of the DUET planner by Gerevini et al. (2008), it is done by calling an instance of an HTN planner based on task insertion called SHOP2 (Nau et al., 2003) to deal with composite actions. Some planners take the integration further by making the decomposition of composite actions into a special step in their refinement process. Such works include the discourse generation oriented DPOCL (Young and Moore, 1994) and the work of Kambhampati et al. (1998) generalizing the practice for decompositional planners.

In our case, we chose a class of hierarchical planners based on Plan Space Planning (PSP) algorithms (Bechon et al., 2014; Dvorak et al., 2014; Bercher et al., 2014) as a reference approach. The main difference here is that the decomposition is integrated into the classical POCL algorithm by only adding new types of flaws. This allows to keep all the flexibility and properties of POCL while adding the expressivity and abstraction capabilities of HTN. We also made an improved planning framework based on the one used by HiPOP to reduce further the number of changes needed to handle composite actions and to increase the efficiency of the resulting implementation.

As stated previously, our goal is to obtain intermediary abstract plans and to evaluate their properties. Another work has already been done on another aspect of those types of plans. The Angelic algorithm by Marthi et al. (2007) exploited the usefulness of such plans in the planning process itself and used them as a heuristic guide. They also proved that, for a given fluent semantics, it is guaranteed that such abstract solutions can be refined into actual solutions. However, the Angelic planner does not address the inherent properties of such abstract plans as approximate solutions and uses a more restrictive totally ordered framework.

Definitions

In order to make the notations used in the paper more understandable we gathered them in table 1. For domain and problem representation, we use a custom knowledge description language that is inspired from RDF Turtle (Beckett and Berners-Lee, 2011) and is based on triples and propositional logic. In that language quantifiers are used to quantify variables `*(x)` (forall x) but can also be simplified with an implicit form : `lost(~)` meaning "*nothing is lost*". For reference the *exclusive quantifier* we introduced (noted `~`) is used for the negation (e.g. `~(lost(_))` for *"something is not lost"*) as well as the symbol for nil. All symbols are defined as they are first used. If a symbol is used as a parameter and is referenced again in the same statement, it becomes a variable.

Domain

The domain specifies the allowed operators that can be used to plan and all the fluents they use as preconditions and effects.

Definition 1 (Domain)**.** A domain is a triplet $\mathcal{D} = \langle E_{\mathcal{D}}, R, A_{\mathcal{D}} \rangle$

- $E_{\mathcal{D}}$ is the set of **domain entities**.
- R is the set of **relations** over $E_{\mathcal{D}}^n$. These relations are akin to n-ary predicates in first order logic.
- $A_{\mathcal{D}}$ is the set of **operators** which are fully lifted *actions*.

[1]Diverse planning aims to find a set of m plans that are distant of d from one another.

Table 1: Our notations are adapted from Ghallab et al. (2004). The symbol $\pm$ shows when the notation has signed variants.

Symbol	Description
$\mathcal{D}, \mathcal{P}$	Planning domain and problem.
$pre(a)$, $eff(a)$	Preconditions and effects of the action a.
$methods(a)$	Methods of the action a.
$\phi^{\pm}(l)$	Signed incidence function for partial order plans. ϕ^{-} gives the source and ϕ^{+} the target step of l. No sign gives a pair corresponding to link l.
$L^{\pm}(a)$	Set of incoming (L^{-}) and outgoing (L^{+}) links of step a. No sign gives all adjacent links.
$a_s \xrightarrow{c} a_t$	Link with source a_s, target a_t and cause c.
$causes(l)$	Gives the causes of a causal link l.
$a_a \succ a_s$	A step a_a is anterior to the step a_s.
A_x^n	Proper actions set of x down n levels.
$lv(x)$	Abstraction level of the entity x.
$a \rhd^{\pm} a'$	Transpose the links of action a onto a'.
$l \downarrow a$	Link l participates in the partial support of step a.
$\pi \Downarrow a$	Plan π fully supports a.
$\ddagger_f\ a$	Subgoal : Fluent f is not supported in step a.
$a_b \boxtimes l$	Threat : Breaker action a_b threatens causal link l.
$a \oplus^m$	Decomposition of composite action a using method m.
$var : exp$	The colon is a separator to be read as "such that".
$[exp]$	Iverson's brackets: 0 if $exp = false$, 1 otherwise.

Example: The example domain in listing 1 is inspired from the kitchen domain of Ramirez and Geffner (2010).

```
take(item) pre (taken(~), ?(item));
     //?(item) is used to make item into
     a variable.
take(item) eff (taken(item));
heat(thing) pre (~(hot(thing)),
     taken(thing));
heat(thing) eff (hot(thing));
pour(thing, into) pre (thing ~(in) into,
     taken(thing));
pour(thing, into) eff (thing in into);
put(utensil) pre (~(placed(utensil)),
     taken(utensil));
put(utensil) eff (placed(utensil),
     ~(taken(utensil)));
infuse(extract, liquid, container) ::
     Action; //Composite action of level 1
make(drink) :: Action; // Level 2
     containing infuse
```

Listing 1: Domain file used in our planner. In order to be concise, the methods are omitted.

Definition 2 (Fluent)**.** A fluent f is a parameterized statement $r(arg_1, arg_2, ..., arg_n)$ where:

- $r \in R$ is a relation/function holding a property of the world.
- $arg_{i \in [1,n]} \in E_{\mathcal{D}}$ are the arguments (possibly quantified).
- $n = |r|$ is the arity of r.

Fluents are signed. Negative fluents are noted $\neg f$ and behave as a logical complement. The quantifiers are affected by the sign of the fluents. We do not use the closed world hypothesis: fluents are only satisfied when another compatible fluent is provided. Sets of fluents have a boolean value that equals the conjunction of all its fluents.

Example: To describe an item not being held, we use the fluent $\neg taken(item)$. If the cup contains water, $in(water, cup)$ is true.

Plan and hierarchical representation

Definition 3 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\pi = (S, L)$, with:

- S the set of **steps** of the plan as vertices. A step is an action belonging in the plan. S must contain an initial step I_π and goal step G_π.
- L the set of **causal links** of the plan as edges. We note $l = a_s \xrightarrow{c} a_t$ the link between its source a_s and its target a_t caused by the set of fluents c. If $c = \emptyset$ then the link is used as an ordering constraint.

In our framework, *ordering constraints* are defined as the transitive cover of causal links over the set of steps. We note ordering constraints: $a_a \succ a_s$, with a_a being *anterior* to its *successor* a_s. Ordering constraints cannot form cycles, meaning that the steps must be different and that the successor cannot also be anterior to its anterior steps: $a_a \neq a_s \wedge a_s \not\succ a_a$. In all plans, the initial and goal steps have their order guaranteed: $I_\pi \succ G\pi \wedge \nexists a_x \in S_\pi : a_x \succ I_\pi \vee G_\pi \succ a_x$. If we need to enforce order, we simply add a link without specifying a cause. The use of graphs and implicit order constraints help to simplify the model while maintaining its properties.

The central notion of planning is operators. Instantiated operators are usually called *actions*. In our framework, actions can be partially instantiated. We use the term action for both lifted and grounded operators.

Definition 4 (Action)**.** An action is a parametrized tuple $a(args) = \langle name, pre, eff, methods \rangle$ where:

- $name$ is the **name** of the action.
- pre and eff are sets of fluents that are respectively the **preconditions and the effects** of the action.
- $methods$ is a set of **methods** (*partial order plans*) that decompose the action into smaller ones. Methods, and the methods of their enclosed actions, cannot contain the parent action.

Example: The precondition of the operator $take(item)$ is simply a single negative fluent noted $\neg taken(item)$ ensuring the variable $item$ is not already taken.

Composite actions are represented using methods. An action without methods is called *atomic*. It is of interest to note the divergence with classical HTN representation here since normally composite actions do not have preconditions nor effects. In our case we insert them into abstract plans.

Input control

In order to verify the input of the domain, the causes of the causal links in the methods are optional. If omitted, the causes are inferred by unifying the preconditions and effects with the same mechanism as in the subgoal resolution in our POCL algorithm. Since we want to guarantee the validity of abstract plans, we need to ensure that user provided plans are solvable. We use the following formula to compute the final preconditions and effects of any composite action a: $pre(a) = \bigcup_{a_s \in L^+(a)} causes(a_s)$ and $eff(a) = \bigcup_{a_s \in L^-(a)} causes(a_s)$. An instance of the classical POCL algorithm is then run on the problem $\mathcal{P}_a = \langle \mathcal{D}, C_\mathcal{P}, a \rangle$ to ensure its coherence. The domain compilation fails if POCL cannot be completed. Since our decomposition hierarchy is acyclic ($a \notin A_a$, see definition 10) nested methods cannot contain their parent action as a step.

Problem

Problem instances are often most simply described by two components: the initial state and the goal.

Definition 5 (Problem)**.** The planning problem is defined as a tuple $\mathcal{P} = \langle \mathcal{D}, C_\mathcal{P}, a_0 \rangle$ where:

- $\mathcal{D}$ is a planning domain.
- $C_\mathcal{P}$ is the set of **problem constants** disjoint from the domain constants.
- a_0 is the **root operator** of the problem which methods are potential solutions of the problem.

Example: We use a simple problem for our example domain. The initial state provides that nothing is ready, taken or hot and all containers are empty (all using quantifiers). The goal is to have tea made. For reference, listing 2 contains the problem instance we use as an example.

```
init eff (hot(~), taken(~), placed(~), ~
    in ~);
goal pre (hot(water), tea in cup, water
    in cup, placed(spoon), placed(cup));
```

Listing 2: Example of a problem instance for the kitchen domain.

The root operator is initialized to $a_0 = \langle "", s_0, s^*, \{\pi_{lv(a_0)}\} \rangle$, with s_0 being the initial state and s^* the goal specification. The method $\pi_{lv(a_0)}$ is a partial order plan with the initial and goal steps linked together via a_0. The initial partial order plan is $\pi_{lv(a_0)} = (\{I, G\}, \{I \xrightarrow{s_0} a_0 \xrightarrow{s^*} G\})$, with $I = \langle "init", \emptyset, s_0, \emptyset \rangle$ and $G = \langle "goal", s^*, \emptyset, \emptyset \rangle$.

Partial Order Causal Links

Our method is based on the classical POCL algorithm. It works by refining a partial plan into a solution by recursively removing all of its flaws.

Definition 6 (Flaws)**.** Flaws have a *proper fluent* f and a causing step often called the *needer* a_n. Flaws in a partial plan are either:

- **Subgoals**, *open conditions* that are yet to be supported by another step a_n often called *provider*. We note subgoals $\ddagger_f\, a_n$.
- **Threats**, caused by steps that can break a causal link with their effects. They are called *breakers* of the threatened link. A step a_b threatens a causal link $l_t = a_p \xrightarrow{f} a_n$ if and only if $\neg f \in eff(a_b) \wedge a_b \not\succ a_p \wedge a_n \not\succ a_b$. Said otherwise, the breaker can cancel an effect of a providing step a_p, before it gets used by its needer a_n. We note threats $a_b \otimes l_t$.

Example: Our initial plan contains two unsupported subgoals: one to make the tea ready and another to put sugar in it. In this case, the needer is the goal step and the proper fluents are each of its preconditions.

These flaws need to be fixed in order for the plan to be valid. In POCL it is done by finding their resolvers.

Definition 7 (Resolvers)**.** Classical resolvers are additional causal links that aim to fix a flaw.

- *For subgoals*, the resolvers are a set of potential causal links containing the proper fluent f in their causes while taking the needer step a_n as their target and a **provider** step a_p as their source.
- *For threats*, we usually consider only two resolvers: **demotion** ($a_b \succ a_p$) and **promotion** ($a_n \succ a_b$) of the breaker relative to the threatened link. We call the added causeless causal link a **guarding** link.

Example: The subgoal for $in(water, cup)$, in our example, can be solved by using the action $pour(water, cup)$ as the source of a causal link carrying the proper fluent as its only cause.

The application of a resolver does not necessarily mean progress. It can have consequences that may require reverting its application in order to respect the backtracking of the POCL algorithm.

Definition 8 (Side effects)**.** Flaws that are caused by the application of a resolver are called *related flaws*. They are inserted into the *agenda*[2] with each application of a resolver:

- *Related subgoals* are all the new open conditions inserted by new steps.
- *Related threats* are the causal links threatened by the insertion of a new step or the deletion of a guarding link.

Flaws can also become irrelevant when a resolver is applied. It is always the case for the targeted flaw, but this can also affect other flaws. Those *invalidated flaws* are removed from the agenda upon detection:

- *Invalidated subgoals* are subgoals satisfied by the new causal links or the removal of their needer.
- *Invalidated threats* happen when the breaker no longer threatens the causal link because the order guards the threatened causal link or either of them have been removed.

[2]An agenda is a flaw container used for the flaw selection of POCL.

Example: Adding the action $pour(water, cup)$ causes a related subgoal for each of the preconditions of the action which are: the cup and the water must be taken and water must not already be in the cup.

In algorithm 1 we present a generic version of POCL inspired by Ghallab et al. (2004).

Algorithm 1 Partial Order Planner

```
 1  function POCL(Agenda $a$, Problem $\mathcal{P}$)
 2      if $a = \varnothing$ then             ▷ Populated agenda needs to be provided
 3          return Success                 ▷ Stops all recursion
 4      Flaw $f \leftarrow$ choose($a$)            ▷ Heuristically chosen flaw
 5      Resolvers $R \leftarrow$ solve($f$, $\mathcal{P}$)
 6      for all $r \in R$ do               ▷ Non-deterministic choice operator
 7          apply($r$, $\pi$)                  ▷ Apply resolver to partial plan
 8          Agenda $a' \leftarrow$ update($a$)
 9          if POCL($a'$, $\mathcal{P}$) = Success then    ▷ Refining recursively
10              return Success
11          revert($r$, $\pi$)         ▷ Failure, undo resolver application
12      $a \leftarrow a \cup \{f\}$                    ▷ Flaw was not resolved
13      return Failure       ▷ Revert to last non-deterministic choice
```

For our version of POCL we follow a refinement procedure that works in several generic steps. In figure 1 we detail the resolution of a subgoal as done in the algorithm 1.

The first is the search for resolver. It is often done in two separate steps : first select the candidates and then check each of them for validity. This is done using the polymorphic function `solve` at line 5.

In the case of subgoals, variable unification is performed to ensure the compatibility of the resolvers. Since this step is time consuming, the operator is instantiated accordingly at this step to factories the computational effort. Composite operators have also all their methods instantiated at this step if they are selected as a candidate.

Then a resolver is picked non-deterministically for applications (this can be heuristically driven). At line 7 the resolver is effectively applied to the current plan. All side effects and invalidations are handled during the update of the agenda at line 8. If a problem occurs, line 11 backtracks and tries other resolvers. If no resolver fits the flaw, the algorithm backtracks to previous resolver choices to explore all the possible plans and ensure completeness.

In definition 8, we mentioned effects that aren't present in classical POCL, namely *negative resolvers*. All classical resolvers only add steps and causal links to the partial plan. Our method needs to remove composite steps and their adjacent links when expanding them.

The Heart of the Method

In this section, we explain how our method combines POCL with HTN planning and how they are used to generate intermediary abstract plans.

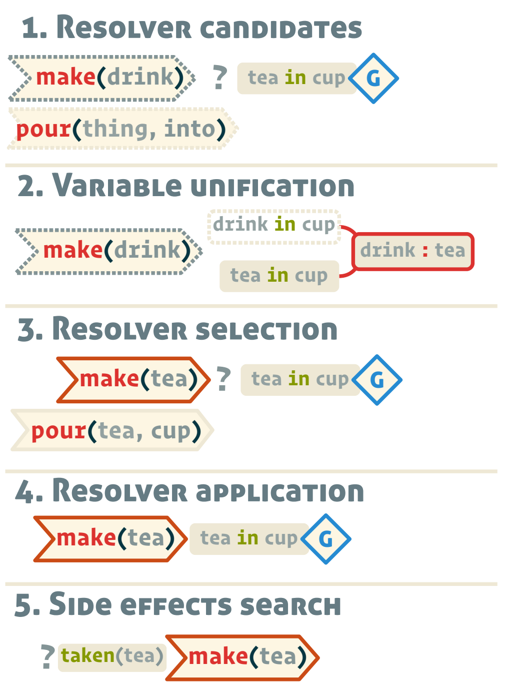

Figure 1: Example of the refinement process for subgoal resolution

Additional Notions

In order to properly introduce the changes made for using HTN domains in POCL, we need to define a few notions.

Transposition is needed to define decomposition.

Definition 9 (Transposition). In order to transpose the causal links of an action a' with the ones of an existing step a in a plan π, we use the following operation :

$$a \rhd^-_\pi a' = \left\{ \phi^-(l) \xrightarrow{causes(l)} a' : l \in L^-_\pi(a) \right\}$$

It is the same with $a' \xrightarrow{causes(l)} \phi^+(l)$ and L^+ for $a \rhd^+ a'$. This supposes that the respective preconditions and effects of a and a' are equivalent. When not signed, the transposition is generalized: $a \rhd a' = a \rhd^- a' \cup a \rhd^+ a'$.

Example: $a \rhd^- a'$ gives all incoming links of a with the target set to a' instead.

Definition 10 (Proper Actions). Proper actions are actions that are "contained" within an entity. We note this notion $A_a = A_a^{lv(a)}$ for an action a. It can be applied to various concepts :

- For a *domain* or a *problem*, $A_{\mathcal{P}} = A_{\mathcal{D}}$.
- For a *plan*, it is $A^0_\pi = S_\pi$.
- For an *action*, it is $A^0_a = \bigcup_{m \in methods(a)} S_m$. Recursively: $A^n_a = \bigcup_{b \in A^0_a} A^{n-1}_b$. For atomic actions, $A_a = \varnothing$.

Example: The proper actions of $make(drink)$ are the actions contained within its methods. The set of extended proper actions adds all proper actions of its single proper composite action $infuse(drink, water, cup)$.

Definition 11 (Abstraction Level)**.** This is a measure of the maximum amount of abstraction an entity can express:[3]

$$lv(x) = \left(\max_{a \in A_x}(lv(a)) + 1\right)[A_x \neq \varnothing]$$

Example: The abstraction level of any atomic action is 0 while it is 2 for the composite action $make(drink)$. The example domain (in listing 1) has an abstraction level of 3.

Abstraction In POCL

The most straightforward way to handle abstraction in regular planners is illustrated by Duet (Gerevini et al., 2008) by managing hierarchical actions separately from a task insertion planner. We chose to add abstraction in POCL in a manner inspired by the work of Bechon et al. (2014) on a planner called HiPOP. The difference between the original HiPOP and our implementation of it is that we focus on the expressivity and the ways flaw selection can be exploited for partial resolution. Our version is lifted at runtime while the original is grounded for optimizations. All mechanisms we have implemented use POCL but with different management of flaws and resolvers. The original algorithm 1 is left untouched.

One of those changes is that resolver selection needs to be altered for subgoals. Indeed, as stated by the authors of HiPOP : the planner must ensure the selection of high-level operators in order to benefit from the hierarchical aspect of the domain, otherwise, adding operators only increases the branching factor. We also need to add a way to deal with composite actions once inserted in the plan to reduce them to their atomic steps.

Definition 12 (Decomposition Flaws)**.** They occur when a partial plan contains a non-atomic step. This step is the needer a_n of the flaw. We note its decomposition $a_n\oplus$.

- *Resolvers*: A decomposition flaw is solved with a **decomposition resolver**. The resolver will replace the needer with one of its instantiated methods $m \in methods(a_n)$ in the plan π. This is done by using transposition such that: $a_n \oplus^m_\pi = \langle S_m \cup (S_\pi \setminus \{a\}), a_n \triangleright^- I_m \cup a_n \triangleright^+ G_m \cup (L_\pi \setminus L_\pi(a_n))\rangle$.
- *Side effects*: A decomposition flaw can be created by the insertion of a composite action in the plan by any resolver and invalidated by its removal :

$$\bigcup_{a_m \in S_m}^{f \in pre(a_m)} \pi' \ddagger_f a_m \bigcup_{a_b \in S_{\pi'}}^{l \in L_{\pi'}} a_b \otimes l \bigcup_{a_c \in S_m}^{lv(a_c) \neq 0} a_c \oplus$$

Example: When adding the step $make(tea)$ in the plan to solve the subgoal that needs tea being made, we also introduce a decomposition flaw that will need this composite step replaced by its method using a decomposition resolver. In order to decompose a composite action into a plan, all existing links are transposed to the initial and goal step of the selected method, while the composite action and its links are removed from the plan. The main differences between HiPOP and HEART in our implementations are the functions of flaw selection and the handling of the results (one plan for HiPOP and a plan per cycle for HEART). In HiPOP, the flaw selection is made by prioritizing the decomposition flaws. Bechon et al. (2014) state that it makes the full resolution faster. However, it also loses opportunities to obtain abstract plans in the process.

[3]We use Iverson brackets here, see notations in table 1.

Cycles

The main focus of our work is toward obtaining **abstract plans** which are plans that are completed while still containing composite actions. In order to do that the flaw selection function will enforce cycles in the planning process.

Definition 13 (Cycle)**.** A cycle is a planning phase defined as a triplet $c = \langle lv(c), agenda, \pi_{lv(c)} \rangle$ where : $lv(c)$ is the maximum abstraction level allowed for flaw selection in the $agenda$ of remaining flaws in partial plan $\pi_{lv(c)}$. The resolvers of subgoals are therefore constrained by the following: $a_p \downarrow_f a_n : lv(a_p) \leq lv(c)$.

During a cycle all decomposition flaws are delayed. Once no more flaws other than decomposition flaws are present in the agenda, the current plan is saved and all remaining decomposition flaws are solved at once before the abstraction level is lowered for the next cycle: $lv(c') = lv(c) - 1$. Each cycle produces a more detailed abstract plan than the one before.

Abstract plans allow the planner to do an approximate form of anytime execution. At any given time the planner is able to return a fully supported plan. Before the first cycle, the plan returned is $\pi_{lv(a_0)}$.

Example: In our case using the method of intent recognition of Sohrabi et al. (2016), we can already use $\pi_{lv(a_0)}$ to find a likely goal explaining an observation (a set of temporally ordered fluents). That can make an early assessment of the probability of each goal of the recognition problem.

For each cycle c, a new plan $\pi_{lv(c)}$ is created as a new method of the root operator a_0. These intermediary plans are not solutions of the problem, nor do they mean that the problem is solvable. In order to find a solution, the HEART planner needs to reach the final cycle c_0 with an abstraction level $lv(c_0) = 0$. However, these plans can be used to derive meaning from the potential solution of the current problem and give a good approximation of the final result before its completion.

Example: In the figure 2, we illustrate the way our problem instance is progressively solved. Before the first cycle c_2, all we have is the root operator and its plan π_3. Then within the first cycle, we select the composite action $make(tea)$ instantiated from the operator $make(drink)$ along with its methods. All related flaws are fixed until all that is left in the agenda is the abstract flaws. We save the partial plan π_2 for this cycle and expand $make(tea)$ into a copy of the current plan π_1 for the next cycle. The solution of the problem will be stored in π_0 once found.

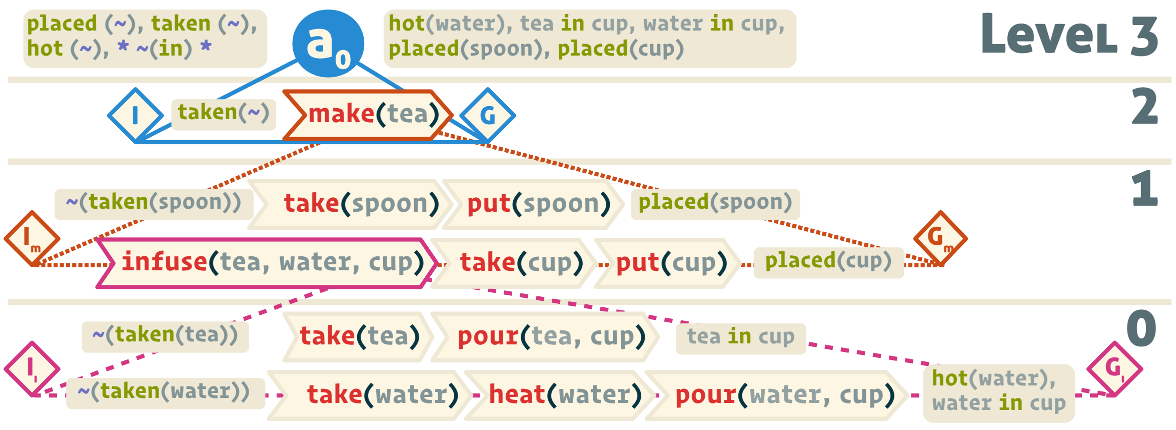

Figure 2: Illustration of how the cyclical approach is applied on the example domain. Atomic actions that are copied from a cycle to the next are omitted.

Results

Theoretical

In this section, we prove several properties of our method and resulting plans : HEART is complete, sound and its abstract plans can always be decomposed into a valid solution.

The completeness and soundness of POCL has been proven in (Penberthy et al., 1992). An interesting property of POCL algorithms is that flaw selection strategies do not impact these properties. Since the only modification of the algorithm is the extension of the classical flaws with a decomposition flaw, all we need to explore, to update the proofs, is the impact of the new resolver. By definition, the resolvers of decomposition flaws will take into account all flaws introduced by its resolution into the refined plan. It can also revert its application properly.

Lemma (Decomposing preserves acyclicity)**.** *The decomposition of a composite action with a valid method in an acyclic plan will result in an acyclic plan. Formely,* $\forall a_s \in S_\pi : a_s \not\succ_\pi a_s \implies \forall a'_s \in S_{a\oplus^m_\pi} : a'_s \not\succ_{a\oplus^m_\pi} a'_s$.

Proof. When decomposing a composite action a with a method m in an existing plan π, we add all steps S_m in the refined plan. Both π and m are guaranteed to be cycle free by definition. We can note that $\forall a_s \in S_m : (\nexists a_t \in S_m : a_s \succ a_t \land \neg f \in \mathit{eff}(a_t)) \implies f \in \mathit{eff}(a)$. Said otherwise, if an action a_s can participate a fluent f to the goal step of the method m then it is necessarily present in the effects of a. Since higher level actions are preferred during the resolver selection, no actions in the methods are already used in the plan when the decomposition happens. This can be noted $\exists a \in \pi \implies S_m \cup S_\pi$ meaning that in the graph formed both partial plans m and π cannot contain the same edges therefore their acyclicity is preserved when inserting one into the other.

□

Lemma (Solved decomposition flaws cannot reoccur)**.** *The application of a decomposition resolver on a plan π, guarantees that $a \notin S_{\pi'}$ for any partial plan refined from π without reverting the application of the resolver.*

Proof. As stated in the definition of the methods (definition 4): $a \notin A_a$. This means that a cannot be introduced in the plan by its decomposition or the decomposition of its proper actions. Indeed, once a is expanded, the level of the following cycle $c_{lv(a)-1}$ prevents a to be selected by subgoal resolvers. It cannot either be contained in the methods of another action that are selected afterward because otherwise following definition 11 its level would be at least $lv(a) + 1$.

□

Lemma (Decomposing to abstraction level 0 guarantees solvability)**.** *Finding a partial plan that contains only decomposition flaws with actions of abstraction level 1, guarantees a solution to the problem.*

Proof. Any method m of a composite action $a : lv(a) = 1$ is by definition a solution of the problem $\mathcal{P}_a = \langle \mathcal{D}, C_\mathcal{P}, a \rangle$. By definition, $a \notin A_a$, and $a \notin A_{a\oplus^m_\pi}$ (meaning that a cannot reoccur after being decomposed). It is also given by definition that the instantiation of the action and its methods are coherent regarding variable constraints (everything is instantiated before selection by the resolvers). Since the plan π only has decomposition flaws and all flaws within m are guaranteed to be solvable, and both are guaranteed to be acyclical by the application of any decomposition $a\oplus^m_\pi$, the plan is solvable.

□

Lemma (Abstract plans guarantee solvability)**.** *Finding a partial plan π that contains only decomposition flaws, guarantees a solution to the problem.*

Proof. Recursively, if we apply the previous proof on higher level plans we note that decomposing at level 2 guarantees a solution since the method of the composite actions are guaranteed to be solvable.

□

From these proofs, we can derive the property of soundness (from the guarantee that the composite action provides its effects from any methods) and completeness (since if a composite action cannot be used, the planner defaults to using any action of the domain).

Experimental

In order to assess its capabilities, HEART was tested on two criteria: quality and complexity. All tests were executed on an Intel® Core™ i7-7700HQ CPU clocked at 2.80GHz. The Java process used only one core and was not limited by time or memory.[4] Each experiment was repeated between 700 and 10 000 times to ensure that variations in speed were not impacting the results.

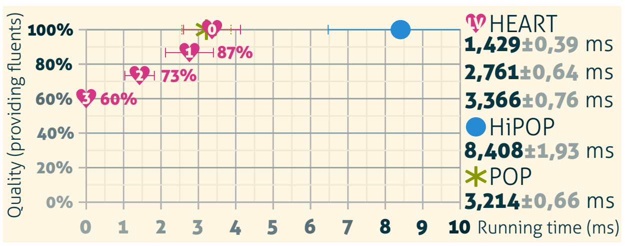

Figure 3: Evolution of the quality with computation time.

Figure 3 shows how the quality is affected by the abstraction in partial plans. The tests are made using our example domain (see listing 1). The quality is measured by counting the number of providing fluents in the plan $\left|\bigcup_{a \in S_\pi} \mathit{eff}(a)\right|$. This metric is actually used to approximate the probability of a goal given observations in intent recognition ($P(G|O)$ with noisy observations, see (Sohrabi et al., 2016)). The percentages are relative to the total number of unique fluents of the complete solution. These results show that in some cases it may be more interesting to plan in a leveled fashion to solve HTN problems. For the first cycle of level 3, the quality of the abstract plan is already of 60%. This is the quality of the exploitation of the plan *before any planning*. With almost three quarters of the final quality and less than half of the complete computation time, the result of the first cycle is a good quality/time compromise.

In the second test, we used generated domains. These domains consist of an action of abstraction level 5. This action has a single method containing a number of actions of level 4. We call this number the width of the domain. All needed actions are built recursively to form a tree shape. Atomic actions only have single fluent effects. The goal is the effect of the higher level action and the initial state is empty. These domains do not contain negative effects. Figure 4 shows the computational profile of HEART for various levels and widths. We note that the behavior of HEART seems to follow an exponential law with the negative exponent of the trend curves seemingly being correlated to the actual width. This means that computing the first cycles has a

[4]The source code of HEART will be available at genn.io/heart

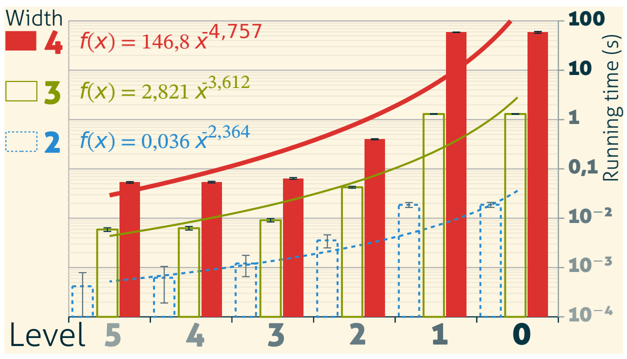

Figure 4: Impact of domain shape on the computation time by levels. The scale of the vertical axis is logarithmic. Equations are the definition of the trend curves.

complexity that is close to being *linear* while computing the last cycles is of the same complexity as classical planning which is at least *P-SPACE* (depending on the expressivity of the domain) (Erol et al., 1995).

Conclusions

In this paper, we have presented a new planner called HEART based on POCL. An updated planning framework fitting the need for such a new approach was proposed. We showed how HEART performs compared to complete planners in terms of speed and quality. While the abstract plans generated during the planning process are not complete solutions, they are exponentially faster to generate while retaining significant quality over the final plans. They are also proof of solvability of the problem. By using these plans, it is possible to find good approximations to intractable problems within tight time constraints.

References

Marco Baioletti, Stefano Marcugini, and Alfredo Milani. 1998. Encoding planning constraints into partial order planning domains. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 608–616. Morgan Kaufmann Publishers Inc. 00013.

Chris L. Baker, Joshua B. Tenenbaum, and Rebecca R. Saxe. 2007. Goal inference as inverse planning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 29. 00064.

Patrick Bechon, Magali Barbier, Guillaume Infantes, Charles Lesire, and Vincent Vidal. 2014. HiPOP: Hierarchical Partial-Order Planning. In *European Starting AI Researcher Symposium*, volume 264, pages 51–60. IOS Press.

David Beckett and Tim Berners-Lee. 2011. Turtle - Terse RDF Triple Language. Technical report, W3C, March.

Pascal Bercher, Shawn Keen, and Susanne Biundo. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Seventh Annual Symposium on Combinatorial Search*.

S. Biundo and B. Schattenberg. 2001. From abstract crisis to concrete relief preliminary report on flexible integration on nonlinear and hierarchical planning. In *Proceedings of the European Conference on Planning*.

Jianxia Chen, Yixin Chen, You Xu, Ruoyun Huang, and Zheng Chen. 2013. A Planning Approach to the Recognition of Multiple Goals. *International Journal of Intelligent Systems*, 28(3):203–216. 00003.

Filip Dvorak, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab. 2014. A flexible ANML actor and planner in robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*.

Kutluhan Erol, James Hendler, and Dana S. Nau. 1994. HTN planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128.

Kutluhan Erol, Dana S. Nau, and Venkatramana S. Subrahmanian. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial intelligence*, 76(1-2):75–88.

Maria Fox. 1997. Natural hierarchical planning using operator decomposition. In *European Conference on Planning*, pages 195–207. Springer.

Maria Fox, Derek Long, and Daniele Magazzeni. 2017. Explainable Planning. In *Proceedings of IJCAI Workshop on Explainable AI*, Melbourne, Australia, August.

Alfonso Gerevini, Ugur Kuter, Dana S. Nau, Alessandro Saetti, and Nathaniel Waisbrot. 2008. Combining Domain-Independent Planning and HTN Planning: The Duet Planner. In *Proceedings of the European Conference on Artificial Intelligence*, volume 18, pages 573–577. 00025.

Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated planning: Theory & practice*. Elsevier, editions. 00002.

Malik Ghallab, Dana Nau, and Paolo Traverso. 2016. *Automated Planning and Acting*. Cambridge University Press, editions. 00058.

Moritz Göbelbecker, Thomas Keller, Patrick Eyerich, Michael Brenner, and Bernhard Nebel. 2010. Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, pages 81–88. AAAI Press, May. 00036.

Subbarao Kambhampati, Amol Mali, and Biplav Srivastava. 1998. Hybrid planning for partially hierarchical domains. In *AAAI/IAAI*, pages 882–888.

Solange Lemai and Félix Ingrand. 2004. Interleaving temporal planning and execution in robotics domains. In *AAAI*, volume 4, pages 617–622. 00117.

Bhaskara Marthi, Stuart J. Russell, and Jason Andrew Wolfe. 2007. Angelic Semantics for High-Level Actions. In *ICAPS*, pages 232–239.

Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404. 00891.

J Scott Penberthy, Daniel S Weld, and others. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. *Kr*, 92:103–114. 00000.

Miquel Ramirez and Hector Geffner. 2009. Plan recognition as planning. In *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling*, volume 19, pages 1778–1783. AAAI Press. 00093.

Miquel Ramirez and Hector Geffner. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence*, volume 24, pages 1121–1126. 00099.

Earl D. Sacerdoti. 1974. Planning in a hierarchy of abstraction spaces. *Artificial intelligence*, 5(2):115–135.

Earl D. Sacerdoti. 1975. The nonlinear nature of plans. Technical report, STANFORD RESEARCH INST MENLO PARK CA.

Bastian Seegebarth, Felix Müller, Bernd Schattenberg, and Susanne Biundo. 2012. Making hybrid plans more clear to human usersa formal approach for generating sound explanations. In *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*, pages 225–233. AAAI Press. 00019.

Shirin Sohrabi, Anton V. Riabov, and Octavian Udrea. 2016. Plan Recognition as Planning Revisited. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 25. 00004.

Daniel S. Weld. 1994. An introduction to least commitment planning. *AI magazine*, 15(4):27. 00775.

R. Michael Young and Johanna D. Moore. 1994. DPOCL: A principled approach to discourse planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 13–20. Association for Computational Linguistics.

HTN Plan Repair Using Unmodified Planning Systems

Daniel Höller and **Pascal Bercher** and **Gregor Behnke** and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{daniel.hoeller, pascal.bercher, gregor.behnke, susanne.biundo}@uni-ulm.de

Abstract

To make planning feasible, planning models abstract from many details of the modeled system. When executing plans in the actual system, the model might be inaccurate in a critical point, and plan execution may fail. There are two options to handle this case: the previous solution can be modified to address the failure (plan repair), or the planning process can be re-started from the new situation (re-planning). In HTN planning, discarding the plan and generating a new one from the novel situation is not easily possible, because the HTN solution criteria make it necessary to take already executed actions into account. Therefore all approaches to repair plans in the literature are based on specialized algorithms. In this paper, we discuss the problem in detail and introduce a novel approach that makes it possible to use unchanged, off-the-shelf HTN planning systems to repair broken HTN plans. That way, no specialized solvers are needed.

1 Introduction

When generating plans that are executed in a real-world system, the planning system needs to be able to deal with execution failures, i.e. with situations during plan execution that are not consistent with the predicted state. Such situations may arise for several reasons. Planning models used for deterministic planning have to abstract from many details of the modeled system and the model might be inaccurate in a critical point. Up to a certain amount of non-determinism in the modeled system, it might also be beneficial to use deterministic planners and deal with execution errors.

Two mechanisms have been developed to deal with such failures: Systems that use *re-planning* discard the original plan and generate a new one from the novel situation. Systems using *plan repair* adapt the original plan so that it can deal with the unforeseen change. In classical planning, the sequence of already executed actions implies no changes other than state transition. The motivation for plan repair in this setting has been *efficiency* (Gerevini and Serina 2000) or *plan stability* (Fox et al. 2006), i.e. finding a new plan that is as similar as possible to the original one.

In hierarchical task network (HTN) planning (Erol, Hendler, and Nau 1996), the hierarchy has wide influence on the set of valid solutions and it makes the formalism also more expressive than classical planning (Höller et al. 2014; 2016). The hierarchy can e.g. enforce that certain actions might only be executed in combination. By simply re-starting the planning process from the new state, those implications are discarded, thus simple re-planning is no option and plans have to be repaired, i.e., the implications have to be taken into account. Several approaches have been proposed in the literature, all of them use special repair algorithms to find the repaired plans.

- In this paper we give an elaborate discussion on the issues that arise when using a re-planning approach that re-starts the planning process from the new state in HTN planning.
- We introduce a novel transformation-based approach that makes it possible to use unchanged, off-the-shelf HTN planning systems to repair broken HTN plans. That way, no specialized solvers are needed.

Next, we introduce HTN planning, specify the formal problem, discuss issues arising when repairing HTN plans, summarize related work, and give our transformation.

2 Formal Framework

This section first introduces HTN planning and specifies the repair problem afterwards.

2.1 HTN Planning

In HTN planning, there are two types of tasks: *primitive* tasks equal classical planning actions, which cause state transitions. *Abstract* tasks describe more abstract behavior. They can not be applied to states directly, but are iteratively split into sub-tasks until all tasks are primitive.

We use the formalism by Höller et al. (2016). Here, a classical planning domain is defined as a tuple $P_c = (L, A, s_0, g, \delta)$, where L is a set of propositional state features, A a set of action names, and $s_0, g \in 2^L$ are the initial state and the goal definition. A state $s \in 2^L$ is a *goal state* if $s \supseteq g$. The tuple $\delta = (prec, add, del)$ defines the preconditions $prec$ as well as the add and delete effects (add, del) of actions, all are functions $f : A \rightarrow 2^L$. An action a is applicable in a state s if and only if $\tau : A \times 2^L \rightarrow \{true, false\}$ with $\tau(a, s) \Leftrightarrow prec(a) \subseteq s$ holds. When an (applicable) action a is applied to a state s, the resulting state is defined as $\gamma : A \times 2^L \rightarrow 2^L$ with $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$. A sequence of actions $(a_0 a_1 \ldots a_l)$ is applicable in a state s_0 if and only if for each a_i it holds that $\tau(a_i, s_i)$, where s_i is for $i > 0$ defined as $s_i = \gamma(a_{i-1}, s_{i-1})$. We will call the state s_{l+1} the resulting state from the application. A sequence of actions $(a_0 a_1 \dots a_l)$ is a solution if and only if it is applicable in the initial state s_0 and results in a goal state.

An HTN planning problem $P = (L, C, A, M, s_0, tn_I, g, \delta)$ extends a classical planning problem by a set of abstract (also called compound) task names C, a set of decomposition methods M, and the tasks that need to be accomplished which are given in the so-called initial task network tn_I. The other elements are equivalent to the classical case. The tasks that need to be done as well as their ordering relation are organized in *task networks*. A task network $tn = (T, \prec, \alpha)$ consists of a set of identifiers T. An identifier is just a unique element that is mapped to an actual task by a function $\alpha : T \to A \cup C$. This way, a single task can be in a network more than once. $\prec : T \times T$ is a set of ordering constraints between the task identifiers. Two task networks are called to be *isomorphic* if they differ solely in their task identifiers. An abstract task can by decomposed by using a decomposition method. A method is a pair (c, tn) of an abstract task $c \in C$ that specifies to which task the method is applicable and a task network tn, the method's subnetwork. When decomposing a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ that includes a task $t \in T_1$ with $\alpha_1(t) = c$ using a method (c, tn), we need an isomorphic copy of the method's subnetwork $tn' = (T', \prec', \alpha')$ with $T_1 \cap T' = \emptyset$. The resulting task network tn_2 is then defined as follows.

$$\begin{aligned} tn_2 =& ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha') \\ \prec_D =& \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup \\ & \{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\} \end{aligned}$$

We will write $tn \to^* tn'$ to denote that a task network tn can be decomposed into a task network tn' by applying an arbitrary number of methods in sequence.

A task network $tn = (T, \prec, \alpha)$ is a solution to a planning problem P if and only if (1) all tasks are primitive, $\forall t \in T : \alpha(t) \in A$, (2) it was obtained via decomposing the initial task network, $tn_I \to^* tn$, (3) there is a sequence $(t_1 t_2 \dots t_n)$ of the task identifiers in T in line with the ordering constraints $\prec$, and the application of $(\alpha(t_1)\alpha(t_2)\dots\alpha(t_n))$ in s_0 results in a goal state.

2.2 Plan Repair Problem in HTN Planning

Next we specify the plan repair problem, i.e., the problem occurring when plan execution fails (that could be solved by plan repair or re-planning), please be aware the ambiguity of this term. A plan repair problem consists of three core elements: The original HTN planning problem P, its original solution plus its already executed prefix, and the execution error, i.e., the state deviation that occurred during executing the prefix of the original solution.

Most HTN approaches that can cope with execution errors do not just rely on the original solution, but also require the modifications that transformed the initial task network into the failed solution. How these modifications look like may depend on the underlying planning system, e.g., whether it is a progression-based system (Nau et al. 2003; Höller et al. 2018a) or a plan-space planner (Bercher, Keen, and Biundo 2014; Dvořák et al. 2014). To have a general definition, we include the so-called decomposition tree (DT) of a given solution tn. A DT is a tree-like representation of performed decompositions. It forms a witness for a decomposition leading to the solution (Geier and Bercher 2011). Its nodes represent tasks; each abstract task is labeled with the method used for decomposing it, the children in the tree correspond to the subtasks of that specific method. All ordering constraints are also represented, such that a DT dt yields the solution tn it represents by restricting the elements of dt to dt's leaf nodes.

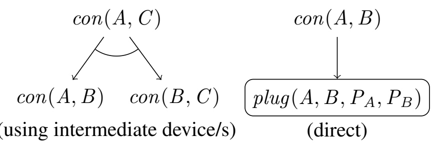

Figure 1: Core methods of an entertainment domain (example from Höller et al. 2018a).

Definition 1 (Plan Repair Problem)**.** *A plan repair problem can now be defined as a tuple $P_r = (P, tn_s, dt, exe, F^+, F^-)$ with the following elements. P is the original planning problem. $tn_s = (T, \prec, \alpha)$ is the failed solution for it, dt the DT as a witness that tn_s is actually a refinement of the original initial task network, and $exe = (t_0, t_1, \dots t_n)$ is the sequence of already executed task identifiers, $t_i \in T$. Finally, the execution failure is represented by the two sets $F^+ \subseteq L$ and $F^- \subseteq L$ indicating the state features that were (not) holding contrary to the expected state after execution the solution prefix exe.*

Though they have been introduced before, we want to make the terms re-planning and plan repair more precise.

Definition 2 (Re-Planning)**.** *The old plan is discarded, a new plan is generated starting from the current state of the system that caused the execution failure.*

Definition 3 (Plan Repair)**.** *The system modifies the non-executed part of the original solution such that it can cope with the unforeseen state change.*

3 About Re-Planning in HTN Planning

In classical planning, a prefix of a plan that has already been executed does not imply any changes to the environment apart from the actions' effects. It is therefore fine to discard the current plan and generate a new one from scratch from the (updated) state of the system. HTN planning provides the domain designer a second means of modeling: the hierarchy. Like preconditions and effects, it can be used to model both physics *or* advice. Figure 1 shows (core parts of) a domain that models the task of assembling an entertainment system. The signal flow is thereby modeled via the hierarchy without using any state features. This can be done by the two given methods. When two devices A and C have to be connected (represented by the task $con(A, C)$), this can be done by using a third intermediate device B, or directly by performing a $plug$ action. That way, devices like a TV or DVD player

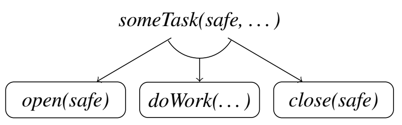

Figure 2: A sketch of a domain containing a pair of actions that have to be executed either both or none.

can be treated equal to cables or adapters and the hierarchy enforces the signal flow. Other things, like which plug fits into which port, or which port is free, can be represented in state. Clearly, this hierarchy represents physics, not advice.

Now imagine a situation where two devices shall be connected and re-planning is performed after half of the connections. Some cables have already been connected to ports and thus both are occupied. When re-planning does not include these circumstances, these cables are just treated as non-free and new cables are used. That way, resources are wasted and in worst case, no solution can be found.

Such situations might be considered during domain design. The domain might include an unplug action, or the recursive connection model can consider plugged cables between devices. However, it *has* to be addressed somehow.

Consider another domain where, for a certain action that causes a safety threat, a second action has to be performed to make the situation safe again, e.g. an action for opening a safe. Every safe that is opened must also be closed eventually. This can easily be modeled as an HTN domain. A sketch for such a domain is given in Figure 2. Though the given domain could also be modeled using some features in classical planning (e.g. by introducing a *closed* state feature and include it in the goal definition for every safe), please be aware that this is not always the case: Consider e.g. that one action needs to be done as many times as a second one. Then, there is no way to ensure it via state, since the state in planning is usually finite. It can, however, be modeled in the more expressive HTN formalism (Höller et al. 2016).

As we have seen in our examples, the hierarchy assures that certain properties hold in every plan and the domain designer might rely on these properties. There are different ways to ensure them:

- The responsibility can be shifted to the domain designer, i.e., the domain must be created in a way that the planning process can be started from any state of the real-world system. This leads to a higher effort for the domain expert and it might also be more error-prone, because the designer has to consider possible re-planning in every intermediate state of the real-world system.
- The reasoning system that triggers planning and provides the planning problem is responsible to incorporate additional tasks to make the system safe again. This shifts the problem to the creator of the execution system. This is even worse, since this might not even be a domain expert, and the execution system has to be domain-specific, i.e., the domain knowledge is split.
- The repair system generates a solution that has the properties assured by the hierarchy. This solution leads to a single model containing the knowledge, the planning domain; and the domain designer does not need to consider every intermediate state of the real system.

Since it represents a fully domain-independent approach, we consider the last solution to be the best. This leads us to a core requirement of a system that solves the plan repair problem: regardless of whether it technically uses plan repair or re-planning, it needs to generate solutions that start with the same prefix of actions that have already been executed. Otherwise, the system potentially discards "physics" that have been modeled via the hierarchy. Therefore we define a solution to the plan repair problem as follows.

Definition 4 (Repaired Plan)**.** *Given a plan repair problem* $P_r = (P, tn_s, dt, exe, F^+, F^-)$ *with* $P = (L, C, A, M, s_0, tn_I, g, \delta)$, $tn_s = (T, \prec, \alpha)$ *and* $exe = (t_0, t_1, \ldots t_n)$*, a repaired plan is a plan that (1) can be executed in* s_0*, (2) is a refinement of* tn_I*, and (3) has a linearization with a prefix equal to* $(\alpha(t_0), \alpha(t_1), \ldots \alpha(t_n))$ *followed by tasks executable despite the unforeseen state change.*

4 HTN Plan Repair: Related Work

Before we survey practical approaches on plan repair in HTN planning, we recap the theoretical properties of the task. Modifying existing HTN solutions (in a way so that the resulting solution lies still in the decomposition hierarchy) is undecidable even for quite simple modifications (Behnke et al. 2016) and even deciding the question whether a given sequence of actions can be generated in a given HTN problem is NP-complete (Behnke, Höller, and Biundo 2015; 2017). Unsurprisingly, the task given here – finding a solution that starts with a given sequence of actions – is indeed undecidable (Behnke, Höller, and Biundo 2015).

We now summarize work concerned with plan repair or re-planning in hierarchical planning in chronological order.

One of the first approaches dealing with execution errors in hierarchical planning is given by Kambhampati and Hendler (1992). It can be characterized as *plan repair*, since they repair the already-found solution with the least number of changes. Though they assume a hierarchical model, the task hierarchy is just advice, i.e., the planning goals are not defined in terms of an initial task network, but as state-based goal. Abstract tasks use preconditions and effects so that they can be inserted as well. They do not base their work upon an execution error, such as an unexpected change of a current situation, but instead assume that the problem description changes, i.e., the initial state and goal description.

Drabble, Dalton, and Tate (1997) introduced algorithms to repair plans in case of action execution failure as well as unexpected world events by modifying the existing plan.

Boella and Damiano (2002) propose a technique that they refer to as re-planning, but the work can be seen as *plan repair* according to our classification. They propose a repair algorithm for a reactive agent architecture. The original problem is given in terms of an initial plan that needs to be refined. Repair starts with a given primitive plan. They take back performed refinements until finding a more abstract plan that can be refined into a new primitive one with an optimal expected utility.

Warfield et al. (2007) propose the RepairSHOP system, which extends the progression-based HTN planner SHOP (Nau et al. 2001) to cope with unexpected changes to the current state. Their *plan repair* approach shows some similarities with the previous one, as they backtrack decompositions up to a point where different options are available that allow a refinement in which the unexpected change does not violate executability. To do this, the authors propose the *goal graph*, which is a representation of the commitments that the planner has already made to find the executed solution.

Bidot, Schattenberg, and Biundo (2008) propose a *plan repair* algorithm to cope with execution errors. The same basic idea has later been described in a more dense way relying on a simplified formalism (Biundo et al. 2011). Their approach also shows similarities to the previous two, as they also start with the failed plan and take planning decisions back, starting with those that introduced failure-associated plan elements, thereby re-using much of the planning effort already done. The already executed plan elements (steps and orderings) are marked with so-called *obligations*, a new flaw class in the underlying flaw-based planning system.

The previous plan repair approach has later been simplified further by Bercher et al. (2014; 2017). Their approach uses obligations to state which plan elements must be part of any solution due to the already-executed prefix. In contrast to the approaches given before, it starts with the initial plan and searches for refinements that achieve the obligations. Technically, it can be regarded *re-planning*, because it starts planning from scratch and from the *original* initial state while ensuring that new solutions start with the already executed prefix. The approach was implemented in the plan-space-based planning system PANDA (Bercher, Keen, and Biundo 2014) and practically in use in the described assembly scenario, but never systematically evaluated empirically.

The most recent approach for HTN *plan repair* that we are aware of is by Barták and Vlk (2017). It focuses on *scheduling*, i.e., the task of allocating resources to actions and scheduling their execution time. In case of an execution error (a changed problem specification), they find another feasible schedule. They perform backjumping (i.e., conflict-directed backtracking) to find repaired solutions.

All these approaches address execution errors by a specialized algorithm. In the next section, we propose a novel approach that solves the problem *without* relying on specialized algorithms. Instead, it encodes the executed plan steps and the execution error into a standard HTN problem, which allows to use standard HTN solvers instead.

5 Plan Repair via Domain Transformation

Technically, the task is similar to our work on *Plan Recognition as Planning* (Höller et al. 2018b). The approach is based on two transformations, one of them enforces HTN plans to start with a prefix of observations.

Let $P_r = (P, tn_s, dt, exe, F^+, F^-)$ be the plan repair problem and $P = (L, C, A, M, s_0, tn_I, g, \delta)$ with $\delta = (prec, add, del)$ the original HTN planning problem, $exe = (a_1, a_2, \ldots, a_m)$ the sequence of already executed actions, and $F^+ \in 2^L$ and $F^- \in 2^L$ the set of the unforeseen positive and negative facts, respectively. Then we define the following HTN planning problem $P' = (L', C', A', M', s'_0, tn'_I, g', \delta')$ with $\delta' = (prec', add', del')$ that solves the plan repair problem.

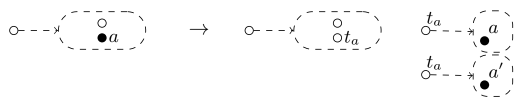

Figure 3: The original method (left) contains the action a that is part of the already executed prefix. This task is replaced by a new abstract task t_a (middle) and two new methods are added that decompose t_a either in a or in a' (right).

First, a sequence of new propositional symbols is introduced that indicate the position of some action in the enforced plan prefix. We denote these facts by l_i with $0 \leq i \leq m$ and $l_i \notin L$ and define the new set of propositional state features as $L' = L \cup \{l_i \mid 0 \leq i \leq m\}$.

For each task a_i with $1 \leq i < m - 1$ in the prefix of executed actions, a new task name a'_i is introduced with $prec'(a'_i) \mapsto prec(a_i) \cup \{l_{i-1}\}$, $add'(a'_i) \mapsto add(a_i) \cup \{l_i\}$ and $del'(a'_i) \mapsto del(a_i) \cup \{l_{i-1}\}$. The last action in the executed prefix a_m needs to have additional effects, it performs the unforeseen state change. $prec'(a'_m) \mapsto prec(a_m) \cup \{l_{m-1}\}$, $add'(a'_m) \mapsto (add(a_m) \setminus F^-) \cup F^+ \cup \{l_m\}$ and $del'(a'_m) \mapsto del(a_m) \cup F^- \cup \{l_{m-1}\}$. The original problem is placed after the prefix, i.e., $\forall a \in A$ holds that $prec'(a) \mapsto prec(a) \cup \{l_m\}$. And the new set of actions is defined as $A' = A \cup \{a'_i \mid 1 \leq i \leq m\}$. To make the first action of the prefix applicable in the initial state, the symbol l_0 is added, i.e., $s'_0 = s_0 \cup \{l_0\}$. To reuse the already executed actions, ensure that every solution starts with the entire prefix, i.e. $g' = g \cup \{l_m\}$.

The newly introduced actions now need to be made reachable via the hierarchy. Since they simulate their duplicates from the prefix of the original plan, the planner should be allowed to place them at the same positions. This can be done by introducing a new abstract task for each action appearing in the prefix, replacing the original action at each position it appears, and adding methods such that this new task may be decomposed into the original or the new action. A schema of the transformation is given in Figure 3. Formally, it is defined in the following way.

$$C' = C \cup \{c'_a \mid a \in A\}, c'_a \notin C \cup A,$$

$$M^c = \{(c, (T, \prec, \alpha')) \mid (c, (T, \prec, \alpha)) \in M\}, \text{ where}$$

$$\forall t \in T \text{ with } \alpha(t) = n \text{ and } \alpha'(t) = \begin{cases} n, & \text{if } n \in C \\ c'_n, & \text{else.} \end{cases}$$

$$M^a = \{(c'_a, (\{t\}, \emptyset, \{t \mapsto a\})) \mid \forall a \in A\},$$

So far the new abstract tasks can only be decomposed into the original action. Now we allow the planner to place the new actions at the respective positions by introducing a new method for every action in $exe = (a_1, a_2, \ldots, a_m)$, decomposing a new abstract task c'_{a_i} into the executed action a_i: $M^{exe} = \{(c'_{a_i}, (\{t\}, \emptyset, \{t \mapsto a'_i\})) \mid a_i \in exe\}$. The set of methods is defined as $M' = M^c \cup M^a \cup M^{exe}$ and all elements of P' have been specified.

Like the approach given by Bercher et al. (2014), our transformation is technically a hybrid between re-planning (the planning process is started from scratch), but the system generates a solution that starts with the executed prefix and incorporates constraints induced by the hierarchy. Since it enforces the properties by using a transformation, the system that generates the actual solution can be a standard HTN planning system. For future work, it might be interesting to adapt the applied planning heuristic to increase plan stability (though this would, again, lead to a specialized system).

6 Conclusion

In this paper we introduced a novel approach to repair broken plans in HTN planning. We elaborated that simply re-starting the planning process is no option since this would discard changes implied by the hierarchical part of the model. Instead, systems need to come up with a new plan that starts with the actions that have already been executed. All systems in the literature tackle the given problem by modifying the applied planning system. We provided a compilation-based approach that enables the use of unchanged HTN planning systems. In future work, we want to empirically evaluate the feasibility of our approach.

Acknowledgments

This work was partly done within the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG).

References

Barták, R., and Vlk, M. 2017. Hierarchical task model forresource failure recovery inproduction scheduling. In *Proc. of the 15th Mexican Int. Conf. on AI (MICAI 2016)*, 362–378. Springer.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2016. Change the plan – How hard can that be? In *Proc. of ICAPS 2016*, 38–46. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. of ICAPS 2015*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (...but is it though?) – Verifying solutions of hierarchical planning problems. In *Proc. of ICAPS 2017*, 20–28. AAAI Press.

Bercher, P.; Biundo, S.; Geier, T.; Hörnle, T.; Nothdurft, F.; Richter, F.; and Schattenberg, B. 2014. Plan, repair, execute, explain – How planning helps to assemble your home theater. In *Proc. of ICAPS 2014*, 386–394. AAAI Press.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2017. *Companion Technology – A Paradigm Shift in Human-Technology Interaction*. Cognitive Technologies. Springer. chapter 5: User-Centered Planning, 79–100.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of SoCS 2014*, 35–43. AAAI Press.

Bidot, J.; Schattenberg, B.; and Biundo, S. 2008. Plan repair in hybrid planning. In *Proc. of the 31st German Conf. on AI (KI 2008)*, 169–176. Springer.

Biundo, S.; Bercher, P.; Geier, T.; Müller, F.; and Schattenberg, B. 2011. Advanced user assistance based on AI planning. *Cognitive Systems Research* 12(3-4):219–236.

Boella, G., and Damiano, R. 2002. A replanning algorithm for a reactive agent architecture. In *Proc. of the 10th Int. Conf. on AI: Methodology, Systems, and Applications (AIMSA 2002)*, 183–192. Springer.

Drabble, B.; Dalton, J.; and Tate, A. 1997. Repairing plans on-the-fly. In *Proc. of the NASA workshop on Planning and Scheduling for Space*, 13–1–13–8.

Dvořák, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and acting with temporal and hierarchical decomposition models. In *Proc. of the 26th IEEE Int. Conf. on Tools with AI (ICTAI 2014)*, 115–121. IEEE.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *Proc. of ICAPS 2006*, 212–221. AAAI Press.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of IJCAI 2011*, 1955–1961. AAAI Press.

Gerevini, A., and Serina, I. 2000. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proc. of the 5th Int. Conf. on AI Planning Systems (AIPS 2000)*, 112–121. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of ECAI 2014*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of ICAPS 2016*, 158–165. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018a. A generic method to guide HTN progression search with classical heuristics. In *Proc. of ICAPS 2018*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018b. Plan and goal recognition as HTN planning. In *Proc. of the AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR 2018)*.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 2001. The SHOP planning system. *AI Magazine* 22(3):91–94.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *JAIR* 20:379–404.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Muñoz-Avila, H. 2007. Adaptation of hierarchical task network plans. In *Proc. of the 20th Int. Florida AI Research Society Conf. (FLAIRS 2007)*, 429–434. AAAI Press.

Programmatic Task Network Planning

Felix Mohr and **Theodor Lettmann** and **Eyke Hüllermeier** and **Marcel Wever**

{firstname.lastname}@upb.de
Paderborn University
Warburgerstrae 100
33098 Paderborn

Abstract

Many planning problems benefit from extensions of classical planning formalisms and modeling techniques, or even require such extensions. Alternatives such as functional STRIPS or planning modulo theories have therefore been proposed in the past. Somewhat surprisingly, corresponding extensions are not available for hierarchical planning, despite their potential usefulness in applications like automated service composition. In this paper, we present programmatic task networks (PTN), a formalism that extends classical HTN planning in three ways. First, we allow both operations and methods to have outputs instead of only inputs. Second, formulas may contain interpreted terms, in particular interpreted predicates, which are evaluated by a theory realized in an external library. Third, PTN planning allows for a second type of tasks, called oracle tasks, which are not resolved by the planner itself but by external libraries. For the purpose of illustration and evaluation, the approach is applied to a real-world use case in the field of automated service composition.

Introduction

It has been known for a long time that defining a planning problem often means to *strategically* organize the search space instead of only describing what is possible. In the initial version of PDDL, this was called the "advise" facet of the definition as opposed to the "physics", which are neutral and only describe what is possible in a domain. And as anticipated, new paradigms such as functional STRIPS (Geffner 2000), numerical planning (Hoffmann 2003), and, more recently, planning modulo theories (Gregory et al. 2012) and planning with the creation of constants (Weber 2009) emerged and significantly improved the language expressivity, the solver efficiency or even both.

Unfortunately, these extensions have not (or only marginally) been transferred to hierarchical planning. One of the main areas of application of hierarchical planning is automated software composition, and specifically that planning domain would strongly benefit from these extensions. More precisely, we are interested in three extensions:

1. *Constant Creation*. Planning actions should be allowed to add new objects to the environment as in (Weber 2009) or (Mohr 2017).
2. *Interpreted Predicates*. Preconditions of methods should be allowed to contain predicates whose truth value can be *evaluated* using background theories (and the state); this is very similar to the PDDL extension proposed in (Gregory et al. 2012).
3. *Oracle Tasks*. For some tasks, it is cumbersome to model its possible refinements by traditional HTN methods, e.g., deciding how to partition a set. Oracle tasks are like primitive tasks that are not linked to operators but to an external function that computes its possible applications *itself*.

In parts, these features have been already implemented in the SHOP2 planning system (Nau et al. 2003). SHOP2 allows for so called *external calls*, which allow to invoke external routines, which, in a way, can be used to interpret predicates and resolve oracle tasks. However, the concrete abilities of SHOP2 in this aspect have not been documented or discussed in scientific literature, so its formal scope is somewhat unclear.

In this paper, we realize these three extensions for hierarchical planning and merge them into a framework we call programmatic task network (PTN) planning. While the first two aspects are rather a transfer of existing classical planning approaches to hierarchical planning, a mechanic like oracle tasks is, to the best of our knowledge, a novelty of our approach.

One question we are particularly after is whether these extensions yield practical advantages. That is, we want to see whether problems can (i) be expressed in a more compact way and (ii) be solved more efficiently in terms of runtime.

To this end, we present a case study in the area of automated machine learning. In fact, the idea of applying hierarchical planning to automated machine learning was our main motivation to extend classical HTN planning. While we do not claim that PTN is relevant for most or even all hierarchical planning domains, the formalism should be relevant also for other problems in the sub-field of automated service composition. Since the case study is a real-world example, a side-contribution of the paper is to demonstrate the application of AI planning to a real use case.

Motivation and Running Example

Automated Machine Learning

Our extension of HTN is mainly motivated by the idea of tackling *automated machine learning* (AutoML) as a planning problem. AutoML is a recent research direction in

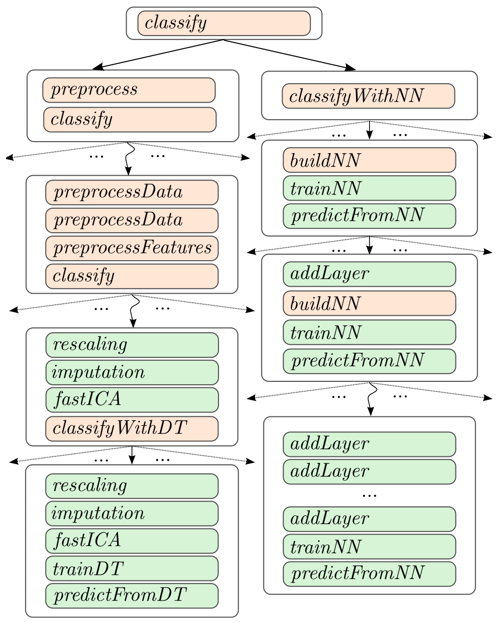

Figure 1: Task networks allow for composing pipelines in a flexible way, and for configuring their elements.

machine learning, which aims at (partly) automating the process of developing a machine learning solution specifically tailored for a concrete data set (Thornton et al. 2013; Feurer et al. 2015). Here, a solution is understood as an "ML pipeline", that is, the selection, composition, and parametrization of algorithms for training a predictor on the data. The latter includes, for example, methods for data pre- and post-processing, induction of a classifier, etc. The pipeline takes the data set as an input and produces a predictor as an output. The quality of a pipeline is measured in terms of the (estimated) generalization performance of the predictor, i.e., its predictive accuracy on new data. This quality may strongly vary between different pipelines.

Our idea is to build an ML pipeline with a hierarchical task network. The point of departure is a single task such as *classify*. This task can be refined recursively by iteratively prepending preprocessing steps and eventually choosing the concrete algorithms and their parametrization. For example, as shown in Figure 1, one could decide to have preprocessing steps before the classifier (left branch), i.e., the node *classify* is replaced by a sequence consisting of two nodes *preprocess* and *classify*. The node *preprocess* could then in turn be refined into two times *preprocessData* and once *preprocessFeatures*. Alternatively, it could be decided that no preprocessing is used (right branch).

Configuring Multi-Class Classifiers

For illustration, we consider a simplified version of the AutoML problem, in which we focus on the configuration of a single element of the pipeline, namely the classification algorithm, while ignoring other steps (such a pre- and post-processing). More specifically, consider the problem of

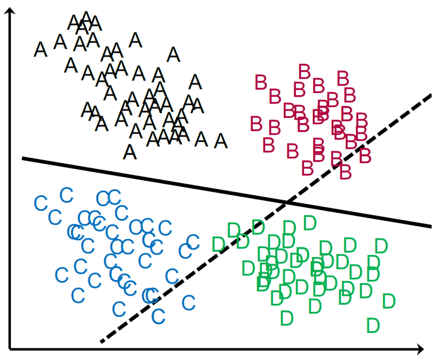

Figure 2: Classification problem with instances as points $x \in \mathbb{R}^2$ and four classes. The meta-class $\{A, B\}$ can easily be separated from $\{C, D\}$ by a linear classifier (solid line). Separating D from $\{A, B, C\}$ is more difficult (dashed line).

training a classifier $h : \mathcal{X} \longrightarrow \mathcal{Y}$, where $\mathcal{X}$ is an instance space (set of data objects) and $\mathcal{Y} = \{y_1, \ldots, y_K\}$ a set of $K > 2$ classes. So-called *decomposition techniques* reduce this problem to a set of binary classification problems, i.e., the training of a set of simple classifiers that can only distinguish between two classes. In so-called *nested dichotomies* (NDs), the reduction is achieved by recursively splitting the set of classes $\mathcal{Y}$ into two subsets (Frank and Kramer 2004).

Formally, a nested dichotomy can be represented by a binary tree, in which every node n is labeled with a set $c(n) \subseteq \mathcal{Y}$ of classes, such that the root is labeled with $\mathcal{Y}$, and $c(n) = c(n_1) \dot\cup c(n_2)$ for every inner node n with successors n_1 and n_2. Every inner node is associated with a binary classifier that seeks to discriminate between the "meta-classes" $c(n_1)$ and $c(n_2)$. At prediction time, a new object to be classified is submitted to the root and, at every inner node, sent to one of the successors by the binary classifier associated with that node; the class assigned is then given by the leaf node reached in the end.

A simple illustration with four classes is given in Fig. 2. Obviously, the dichotomy $((A, B), (C, D))$ would be a good choice for this problem, since all classification problems involved ($\{A, B\}$ versus $\{C, D\}$, A versus B, C versus D) can be solved quite accurately with a simple linear classifier. The dichotomy $((A, D), (B, C))$, on the other hand, would lead to rather poor performance, because the classifier in the root will make many mistakes ($\{A, D\}$ cannot easily be separated from $\{B, C\}$). The dichotomy $(A, (B, (C, D)))$ will produce a mediocre result.

In this paper, we configure nested dichotomies using hierarchical planning, assuming that the *base learner* for solving binary problems is a linear support vector machine. Even this reduced configuration problem is rather challenging. In fact, the problem of finding a dichotomy that is optimal for a given set of data and for a fixed base learner comes down to searching the space of all dichotomies, and the size of this space is $(2n - 3)!!$ for n classes (Frank and Kramer 2004).[1]

[1]Here, !! is the double factorial, not taking the factorial twice.

Configuring nested dichotomies hierarchically seems natural: Starting at the root, the splits are configured iteratively until every leaf node is labeled with exactly one class. In other words, a "complex" problem such as training a classifier for classes $\{A, B, C, D\}$ is (recursively) refined into simpler problems, such as training a classifier for classes $\{A, B\}$ and training a classifier for classes $\{C, D\}$ (and appropriately combining the two). Each of these problems again defines a classification task, which is solved in the same way (even though other techniques could be applied in principle).

The Baseline: The Classical HTN Formalization

We now explain how the configuration of such NDs can be encoded as a classical hierarchical planning problem. The formalization ensures that each ND is constructed exactly once. We need three operators, corresponding to primitive tasks:

1. $\mathit{init}(n, x, lc, rc, nc)$
   Pre: $x \in n \wedge \bullet(lc) \wedge \tau(lc, rc) \wedge \tau(lc, nc)$
   Post: $\bigwedge$
   $\mathit{true} \rightarrow x \in rc \wedge bst(x, rc) \wedge sst(x, rc)$
   $\forall x_n : x_n \in n \wedge x_n \neq x \rightarrow x_n \in lc$
   $\forall x_2, x_o : x \neq x_2 \wedge x_2 \in n \wedge sst(x, n) \wedge (x_o \notin n \vee x_o > x_2) \rightarrow sst(x_2, lc)$
   $\forall x_s : sst(x_s, n) \wedge x_s \neq x \rightarrow sst(x, lc)$
   $\neg \bullet (lc) \wedge \bullet(nc)$
2. $\mathit{shift}(y, x, l, r)$
   Pre: $x \in l \wedge bst(y, r)$
   Post: $x \in r \wedge bst(x, r) \wedge \neg x \in l \wedge \neg bst(y, r)$
3. $\mathit{close}(l, lw, r, rw)$
   Pre: $lw \in l \wedge rw \in r$
   Post: $\emptyset$

Intuitively, the idea behind these operators is to split up the labels of a node until every leaf node is labeled with a single class. A node is refined by creating two child nodes (via the init operator), where initially all classes except one (x) of the parent are in the left child. Then, we can use the shift operator to move single classes from the left to the right child. The predicates bst and sst are used to memorize the biggest and smallest elements of nodes, which is necessary to avoid mirroring NDs, i.e. one separating A, B from C, D and the other C, D from A, B The close operator can be used to guarantee the existence of at least one class in each of the children, which are the "witnesses" lw and rw; this guarantess soundness of solutions.

The relatively complicated notation of the $\bullet$ and τ predicates is to efficiently simulate the creation of objects. The idea is that there is a counter for the next newly created constant, which is shifted whenever an object is "created"; the state of this counter is maintained with $\bullet$. The initial state then needs to contain some successor chain $\tau(c_0, c_1), .., \tau(c_{n-1}, c_n)$ that indicates the order in which the constants are created. Here, n iconstants may be created (the $n + 1$-th constant cannot be created, because no successor is known for it). Hence, it is actually possible to simulate the creation of objects with the only limitation that some bound n for the number of such objects needs to be set. Previous approaches for simulating output creation (Klusch, Gerber, and Schmidt 2005) have used a different, simpler, encoding, which leads to a blow-up of the search space as analyzed in our experimental evaluation section.

While one may object that outputs are then only syntactical sugar, we would argue that a native support for outputs is quite desirable in both the problem formalization and the implementation of tools. On the theoretic side, allowing for outputs is naturally a good thing because this constitutes a specific planning problem, which is generally undecidable even without hierarchies (Hoffmann et al. 2009). On the practical side, planning is precisely about offering syntax (and semantics) to simplify the specification of a special kind of search problem. There are planning domains, in particular software configuration, where outputs are first-class citizens. In HTN planning, simulating the constant creation not only complicates the description of operations but also propagates to methods as can be seen below. Hence, outputs alone may not justify an *implementation* of an entirely new planner but motivate the support of outputs as part of the problem description.

We need two tasks with five methods to complete the specification. The first task is $\mathit{refine}(n)$, which means that the classes of node n shall be split up somehow. The second task is $\mathit{config}(l, r)$, which means that classes are to be moved from the left to the right child of some node. There are three methods for $\mathit{refine}(n)$:

1. $\mathit{finalSplit}(n, x, y, l, r, s)$
   Pre: $x \in n, y \in n, y > x, \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, lc, rc, y)$
2. $\mathit{isolatingSplit}(n, x, l, r, s)$
   Pre: $x \in n, \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, l, r, y) \rightarrow \mathit{refine}(l)$
3. $\mathit{doubleSplit}(n, x, y, l, r, s)$
   Pre: $x \in n, y \in n, y > x, \neg sst(x, n), \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, l, r, y) \rightarrow \mathit{shift}(y, x, l, r) \rightarrow \mathit{config}(l, r) \rightarrow \mathit{refine}(l) \rightarrow \mathit{refine}(r)$

There are two methods for $\mathit{config}(l, r)$, which are

1. $\mathit{shiftElementAndConfigure}(l, r, x, y)$
   Pre: $x \in l, bst(y, r), x > y$
   TN: $shift(x, y, l, r) \rightarrow config(l, r)$
2. $\mathit{closeSetup}(l, lw, r, rw)$
   Pre: $lw \in l, rw \in r$
   TN: $\mathit{close}(l, lw, r, rw)$

The initial task network is then $\{\mathit{refine}(\mathit{root})\}$, where the initial state s_0 defines root and the ordering of classes. That is, $s_0 = \varphi(C) \wedge \bigwedge_{x \in C}(x \in \mathit{root})$, where C is the set of classes and φ maps C to an arbitrary explicit total order of items of C, e.g., the lexicographical order. The latter one is important to maintain the bst and sst predicates.

PTN Planning Formalism

Basic Planning Elements

As for any planning formalism, our basis is a logic language $\mathcal{L}$ and planning operators defined in terms of $\mathcal{L}$. The language $\mathcal{L}$ has first-order logic capacities, i.e., it defines an infinite set of variable names, constant names, predicate names, function names and quantifiers and connectors to build formulas. A *state* is a set of ground literals; i.e., it does not contain unquantified variable symbols. We do *not* adopt the closed-world assumption.

Like in planning modulo theories (Gregory et al. 2012), constants, functions, and a subset of the predicates of $\mathcal{L}$ are taken from a *theory*. A theory $\mathcal{T}$ defines constants, functions, and predicates and how these are to be interpreted. Predicates not contained in $\mathcal{T}$ behave like normal predicates in classical planning. That is, $\mathcal{L}$ consists of the elements of $\mathcal{T}$ together with uninterpreted predicates and constants. In the formalism, we use $\mathcal{T}$ as a formula itself.

An *operator* is a tuple $\langle name_o, I_o, O_o, P_o, E_o^+, E_o^- \rangle$ where $name_o$ is a name, I_o and O_o are parameter names described inputs and outputs, P_o is a formula from $\mathcal{L}$ constituting its preconditions and E_o^+ and E_o^- are sets of conditional statements $\alpha \rightarrow \beta$ where α is a formula over $\mathcal{L}$ conditioning the actual effect β, which is a set of literals from $\mathcal{L}$ to be added or removed. Free variables in P_o must be in I_o and free variables in E_o^+ and E_o^- must be in $I_o \cup O_o$.

The semantics of the planning domain are as follows. An *action* is an operator whose input and output variables have been replaced by constants; we denote P_a, E_a^+, and E_a^- as the respectively replaced preconditions and effects. An action a is *applicable* in a state s under theory $\mathcal{T}$ iff $s, \mathcal{T} \models P_a$ and if none of the output parameters of a is contained in s. Applying action a to state s changes the state in that, for all $\alpha \rightarrow \beta \in E_a^+$, β is added to s if $s, \mathcal{T} \models \alpha$; analogously, β is removed if such a rule is contained in E_a^-. A *plan* for state s_0 is a list of actions $\langle a_0, .., a_n \rangle$ where a_i is applicable and applied to s_i; here, s_{i+1} is obtained by applying a_i to s_i.

To summarize, the main difference in the basic planning formalism to classical planning is that operators have explicit outputs and that some predicates are not only evaluated from the state itself but the state together with some theory. Having theories available to evaluate expressions, predicates in the preconditions and effects may also contain terms others than simple variables. None of the two aspects is new by itself since output parameters have been considered in automated service composition previously (Weber 2009), and interpreted predicates have been considered prior to planning modulo theories (Gregory et al. 2012) through the notion of functional STRIPS (Geffner 2000).

Programmatic Task Networks

On top of this basic planning formalism, we now build a hierarchical model (Alford et al. 2016). A task network (HTN) is a partially ordered set T of tasks. A task $t(v_0, .., v_n)$ is a name with a list of parameters, which are variables or constants from $\mathcal{L}$. A task named by an operator is called *primitive*, otherwise it is *complex*. A task whose parameters are constants is ground.

The goal of HTN planning is to derive a plan for a given initial state and task network. That is, instead of reaching a goal state from the initial state (as in classical planning), we iteratively *refine* a given partial solution (the task network) until only primitive tasks are left.

While primitive tasks are realized canonically by an operation, complex tasks need to be decomposed by *methods*. A method $m = \langle name_m, t_m, I_m, O_m, P_m, T_m \rangle$ consists of its name, the (non-primitive) task t_m it refines, the input and output parameters I_m and O_m, a logic formula $P_m \in \mathcal{L}$ that constitutes the method's precondition, and a task network T_m that realizes the decomposition. The preconditions may, just as in the case of operations, contain interpreted predicates and functional symbols from the theory $\mathcal{T}$.

An method instantiation m is a method where inputs and outputs have been replaced by planning constants. m is *applicable* in a state s under theory $\mathcal{T}$ iff $s, \mathcal{T} \models P_m$ and if none of the output parameters of m is contained in s.

Leaving apart the different outputs of operations and methods and the functional elements in formulas, the definition of a PTN planning problem is analogous to the one of classical HTN planning. That is, a PTN planning problem is a tuple $\langle O, M, s_0, N \rangle$ where O is a set of operations as above, M is a set of methods, s_0 is the initial state, and N is a task network. The conditions for a plan $\pi = \langle a_1, .., a_n \rangle$ that is applicable in s_0 being a *solution* to a PTN problem $\langle O, M, s_0, N \rangle$ are inductive based on three cases:

1. N is empty. π is a solution if it is empty
2. N has a primitive task t without predecessor in N. π is a solution if a_1 realizes t and is applicable in s_0 and if $\langle a_2, .., a_n \rangle$ is a solution to $\langle O, M, \tau(s_0, a_1), N \setminus \{t\} \rangle$.
3. N has a complex task t without predecessor in N. π is a solution if there is an instantiation $\widehat{m}$ of a method $m \in M$ that is applicable in s_0 yielding a refined network N', and π is a solution to $\langle O, M, s_0, N' \rangle$.

Note that these cases are not mutually exclusive unless N is totally ordered.

The above HTN formalization extends classical HTN planning by object creation and interpreted predicates.

In PTN, we allow a third type of tasks we call *oracle* tasks. An oracle task t is a (primitive or complex) task that is associated with *functions* φ_t that generate *sets* of solutions (in the spirit of the above definition of a solution) to the subproblem $\langle O, M, s, \{t\} \rangle$. A programmatic task network is a hierarchical task network that may contain oracle tasks.

The notion of oracle tasks is an entirely algorithmic one and does not affect the semantic of the planning problem. The idea of oracle tasks is that the planner does not solve them by himself but *outsources* the computation of solutions to its oracle functions. In a sense, oracle tasks play the role of "complex" primitive tasks. They are primitive, because they are ground to actions within one planning step, but they are also complex, because they are not necessarily replaced by a single action but a sequence. However, with respect to the definition of a solution, oracle tasks are simply treated as primitive or complex, so whether a task is oracle does not affect the set of solutions to it.

Discussion

Prior to proceeding, we would like to discuss two aspect of PTN. First, what is the relation between using oracle tasks and interleaving planning and execution? Second, is using both interpreted predicates and oracles redundant?

The main difference between using oracle tasks and interleaving planning and execution is that the latter one is usually done to determine the successor state resulting from a *given* action. That is, the action that is going to be executed is *fixed* by the planner. Oracle tasks, in contrast, actually *outsource* a part of the planning (and decision) logic itself to the function bound to it.

The relation between interpreted predicates and oracles is complementary. On one hand, interpreted predicates are useful to ease the formalization process of a planning problem. On the other hand, oracle tasks aim at shortcutting some parts of the planning process and possibly prune alternatives. Of course, one can also achieve the same pruning effect using interpreted predicates, and this can sometimes make sense. However, we simply see the two concepts as used for different purposes and with different times of coming into action: Interpreted predicates enrich the formalization language, and oracles decrease the runtime of the planner. We also consider this aspect in our experimental evaluation.

Describing PTN Planning Problems

Unfortunately, there is no standard language to describe hierarchical planning problems as PDDL is for classical planning. One attempt to create such a language was made with ANML (Smith, Frank, and Cushing 2008), but it has not evolved to a standard. All existing HTN planners use their own format, so there is no commonly agreed point of reference which can serve as a basis for our extension. In a sense, the SHOP2 planner has created some kind of implicit proprietary standard (Nau et al. 2003). As a consequence, describing the way how PTN planning problems are described would come down to explaining the input syntax for our specific planner.

Hence, our format is proprietary, and we rather refer the reader to the technical documentation of the planner, which formally describes the syntax for describing the planning problems. In fact, the description language only puts a specific syntax for the formal items discussed above. The whole planning problem is defined in just one file with several sections for *types*, *constants*, *operations*, *methods*, and *oracles* respectively. There is no added value in describing this syntax in detail at this point.

However, we briefly want to discuss the definition of interpreted predicates and oracles since this is something technically new. As in (Gregory et al. 2012), interpreted terms are described in an extra file, one for each theory. Predicates that do not occur in any of these files are supposed to be not interpreted. Oracles are described by 6-tuples as follows:

```
[Oracles]
rpnd; refineND(n,lc,rc); n; lc,rc; card(n) > 1; rpnd.sh
cbnd; refineND(n,lc,rc); n; lc,rc; card(n) > 1; cbnd.sh
```

In this notation—where fields are separated by the ; symbol—, the first entry defines the name of the oracle, the second one the task addressed by the oracle followed by the input and output parameters, the precondition, and the external library that will be called to conduct the refinement. Note that our implementation is in Java and external libraries are either executable by the used operating system and invoked in a new process or Java classes implementing a specific interface, which, of course, is more performant.

We require that interpreted predicates are not only associated with an evaluation function but also with a *ground truth* function. For a given state, the ground truth function computes *all* possible groundings to objects of the state for which it evaluates to true. That is, a predicate cannot only be evaluated for a fixed ground parameters but they can even be *queried* for valid groundings. An example where this becomes important is the $ssubset$ predicate used in the following formalization; this predicate simply realizes the strict subset relation. The planner has not even information about possible *candidates* s for which it should evaluate $ssubsets(s, p)$, but the underlying set theory can *inspect* the object p in the state and generate objects representing the possible subsets.

In practice, it is not necessary to define methods for complex oracle tasks. The planner will not treat oracle tasks itself, so there is no reason to formalize the methods that can be used to refine it for the planner. In any case, the external libraries are specifically designed for a particular planning problem and usually only create valid solutions by construction. Since those libraries usually do not apply a planning algorithm themselves, a description of the available methods (or even the whole planning domain) can be omitted unless the libraries explicitly require those for whatever reason. Since PMT does not verify the correctness of the oracles' answers using the existing methods but simply trusts that they are correct, the formalization is optional and can be rather seen as a documentation.

We now explain how the nested dichotomy creation problem can be formalized as a PTN problem. We only need one primitive task (and operation) and one complex task (with two methods). The primitive task and its corresponding operation are responsible for configuring a specific split.

$config(p, s; lc, rc)$
Pre: $\emptyset$
Post: $\forall x : x \in s \rightarrow x \in lc, \forall x : x \in p \wedge x \notin s \rightarrow x \in rc$

The operation has two inputs and two outputs. The first input p is the node that is to be refined and the second one, s, corresponds to a specification of a subset of the elements of s that will appear in the left child. The outputs lc and rc are the data containers for the left and right child node respectively. Note that we do not need any precondition, because this action will only be used in a plan if other (method) preconditions were checked before. Since those method preconditions are sufficient, there is no need to formalize the actual preconditions of the action again. This effect of "shifted" preconditions is not special to our example but is common in HTN planning.

In addition to this operation, we need one complex task $refine(n)$ for which we have two methods:

1. $doRefine(n, s, lc, rc)$
   Pre: $\underline{ssubset}(s, p) \ \wedge \ !\underline{empty}(s) \wedge \underline{min}(s) = \underline{min}(p)$
   TN: $config(n, s, lc, rc) \rightarrow refine(lc) \rightarrow refine(rc)$
2. $closeNode(n)$
   Pre: $card(n) = 1$
   TN: $\emptyset$

The first method is to conduct an actual refinement of a node where the second is responsible only to detect that a node has already been refined to the end and can be closed. Once again, the comparison of the minimum element is needed to avoid so called mirrored dichotomies that are actually identical modulo switching the left and the right child of a node (see above formalization).

The preconditions of the methods only contain interpreted terms. All predicates are from a standard set theory as in (Gregory et al. 2012), so the dichotomy problem doesn't require a special theory. Note that, for the case of the first method, the strict subset condition plus the requirement that s is not empty implicitly requires that $card(p) \geq 2$.

In PTN-Plan, the complex task will be resolved using an oracle. Consider the precondition predicate $ssubset(s, p)$ and suppose there are no oracles for the task. When determining the applicable groundings of the operation, the planner must branch over all possible subsets s of p, i.e. $2^{|p|} - 1$ many candidates. This is usually infeasible even for very small p, because the planner must consider this exponential number of candidates not only once but also subsequently when analyzing possible successor nodes. Hence, PTN-Plan outsources the task grounding to an oracle task, which only produces a small number of these candidates. In our evaluation, we consider both cases to illustrate this effect.

A PTN Planner

We adopt a modification of forward decomposition (Ghallab, Nau, and Traverso 2004). In a nutshell, a rest problem in forward decomposition is a state together with a task network. Of course, initially, this is the initial state s_0 and the initially given task network N. Forward decomposition means to take one of the tasks in N that have no predecessors and resolve it either to an operation (if primitive) or to a new task network (if complex). Our planner, PTN-Plan, is written in Java. The implementation is available for public [2].

The classical forward decomposition algorithm must be modified in three ways. We discuss these modifications in detail in the subsequent sections.

Treating Output Variables

Even though outputs are motivated by operation outputs, the point where they become relevant in the algorithm are *methods*. While the constants are actually created by some action, methods need to talk over those outputs in order to establish a reasonable data flow in the task network they induce. For example, if we have a task `refineND(nd)` where `nd` is an object representing a nested dichotomy, we may have a method `configureAndRefineRecursively(nd,lc,rc)` with an induced totally ordered task network `crtAndConfig(nd,lc,rc) -> refineND(lc) -> refineND(rc)`, which is supposed to create two subsequent dichotomies `lc` and `rc` and distribute the elements of `nd` over them. So in fact, it is already clear at the method level that `lc` and `rc` will be produced elements and are not available yet.

[2]URL hidden during review phase

PTN-Plan stores the outputs of a method in opaque *data containers*. With respect to the planning formalism, data containers are nothing special but ordinary planning constants. Intuitively, data containers are what variable *names* are in typical imperative programming. That is, the container object itself is rather a reference to real semantic object than the object itself. PTN-Plan maintains a counter of newly created objects and labels them `newVar1`, `newVar2`, ... For this reason, it is forbidden to use constants with such a name in the problem description in order to avoid confusion.

In particular in the presence of theories, one may be interested in "complex" objects. In planning modulo theories, planning constants are not only some names but actually string representations of more complex objects such as a set. For example, the string "{`a, b, c`}" could be a planning constant with an intended meaning, which is obviously not known to the planner but only to the theory.

Even though PTN-Plan uses names for output objects instead of serializations according to some theory, more precise information about the container may become available later. The concrete value stored of data container, e.g., "{`a, b, c`}" will be determined by an action but usually not the method instance itself. So at the time of determining the method instance itself, it is not possible to say anything about the contents of a data container. But this is also not a problem, because the content description can be easily added using equality. For example, an operation can have an effect saying `o1 = union(i1,i2)` where `union` is a term from the set theory and `i1`,`i2`, and `o1` are inputs and outputs. Since `i1` and `i2` are known, the concrete serialization can be computed using the theory libraries as in (Gregory et al. 2012).

Treating Interpreted Terms

The point where interpreted terms become relevant to PTN-Plan is when it determines the method instantiations or actions that are applicable in a state. Both were defined to be applicable if their preconditions are satisfied in the state module the underlying theory $\mathcal{T}$.

Since the evaluation of interpreted predicates is potentially costly, PTN-Plan first evaluates the "normal" predicates. This process already implies a binding of most (and often all) of the variables of a method or operation, and the interpreted predicate has only to be checked for given parameters instead of determining for which of a given set of possible parameters it holds.

Unlike in planning modulo theories (Gregory et al. 2012), PTN-Plan does not evaluate interpreted terms. That is, PTN-Plan only distinguishes between predicates and terms but does not make a difference between primitive terms (constants) and complex ones (possibly nested expressions). In any case, both are simply string representations encoding some element of the respective theory and need to be decoded by the external library. At this point, we do not see any reason to have two different representations for the same constant within the planning calculus. Of course, PTN-Plan can be extended in this regard if we observe that collapsing a term to a simpler constant is useful in terms of runtime.

Technically, PTN-Plan supports the evaluation of interpreted predicates in two ways. First, PTN-Plan comes with a Java interface which can be implemented by a Java class used to evaluate a predicate. This is the most performant solution, because no new process needs to be spawned for the evaluation. For compatibility with external libraries, however, PTN-Plan also supports the call of stand-alone executable files. In that case, PTN-Plan expects the result of the evaluation (and nothing else) to be returned on the standard output stream; clearly, this variant is much slower.

Treating Oracle Tasks

When selecting an oracle task, PTN-Plan calls the respective oracle library and blocks until a set of solutions for the task arrives. As for interpreted terms, PTN-Plan supports Java oracle classes that implement a specific interface or external libraries that return the set of solutions (and nothing else) in a specific format over the output stream.

The oracle library is invoked with the reduced rest problem as its main parameter. The reduced rest problem is defined by the current state and the oracle task as the only task; the subsequent tasks are irrelevant.

Once the solutions have arrived, PTN-Plan creates one successor for each of the sub-solutions. The rest problem of those successors is the state that results from applying the respective sub-solution to the previous state, and the task network is simply the one of the previous rest problem without the resolved oracle task.

A subtle twist that has not been discussed so far is the fact that the oracle library may want to conduct an informed search, too. That is, PTN-Plan adopts a best-first search and uses some domain-specific source of information to compute the f-values of the nodes, and that source of information should be also available to the oracle libraries. However, this is no problem, because the common source of information can be stored as a resource, e.g., a file name, in a constant of the planning state. The oracle can then inspect that constant and acquire the desired information. In particular, no additional channel of communication is required.

A Brief Analysis of PTN-Plan

Correctness and Completeness

Assuming the correctness of solutions returned by oracles, the correctness of PTN-Plan is straight forward. The overall correctness of a solution π for the problem $\langle O, M, s_0, N\rangle$ follows from induction over the solution length n. PTN-Plan only returns an empty solution ($n = 0$) if $N = \emptyset$, which is correct. For $n > 0$, the first action a_1 of the solution is either inserted individually as the result of resolving a primitive task, or it is part of a sub-solution $\langle a_1, .., a_k\rangle$ generated by an oracle for a (complex) oracle task. The first case only occurs if PTN-Plan chose a primitive task $t \in N$ realized by a_1 and t is not preceeded by any other task in N; PTN-Plan only chooses a_1 if it is applicable. The second case only occurs if N has an oracle task not preceeded by any other task in N; the sub-solution $\langle a_1, ..a_k\rangle$ was then created by an oracle and is correct by assumption. In any case, the length of the solution to the rest problem is smaller than n; so the correctness of PTN-Plan follows from induction.

PTN-Plan is complete for problems without oracles or for oracles that create all solutions that exist for a single oracle task. This follows again by induction over the length of solution $\pi = \langle a_1, .., a_n\rangle$ for a problem $\langle O, M, s_0, N\rangle$. Three cases are possible. First, there is a primitive non-oracle task $t \in N$ realized by a_1 without predecessor in N; PTN-Plan considers a_1 as a possible refinement. Second, there is an oracle task $t \in N$ without predecessor to which $\langle a_1, .., a_l\rangle$ is a solution. Then, by assuming that oracles create all solutions, PTN-Plan obtains $\langle a_1, .., a_l\rangle$ from some oracle. Third, there is a complex non-oracle task $t \in N$ without predecessor in N. There is a refinement N' of N obtainable by the application of applicable method instantiations $m_1, .., m_k$ such that π is still a solution and that has no complex non-oracle task without predecessor in N'. But PTN-Plan considers these method instantiations in that order such that eventually one of the first two cases applies. In any case, PTN-Plan will eventually arive at a sub-solution $\langle a_1, .., a_l\rangle$ and a problem for which a solution $\langle a_{l+1}, .., a_n\rangle$ exists and which is found by the induction hypothesis.

However, since it is precisely the purpose of oracles to prune parts of the search space that seem irrelevant to them, PTN-Plan is not complete in general. That is, there are solutions that are not contained in the search graph of PTN-Plan. By the above analysis on completeness, this is the case if and only if the set of solutions created by the union of oracles defined for a task is a strict subset of the actual solution set. Consequently, oracle tasks can be used to trade completeness for search efficiency.

Heuristic Search

PTN-Plan adopts a best-first-epsilon algorithm to conduct the search over the graph induced by the planning problem. In our domains, A* is typically not applicable, because the criterion that is subject to optimization is not the plan length but some other qualitative property of the solutions that often does not decompose in an additive way over the edges of the search graph. For example, we cannot estimate the prediction accuracy of a nested dichotomy in an additive way over the search path.

As most hierarchical planners, PTN-Plan is not equipped with a built-in heuristic. We are aware of two planners with a built-in heuristic we are aware of. One is PANDA (Bercher and others 2015), and the other is Hierarchical Goal Network Planning (HGN) (Shivashankar et al. 2012; 2013; Shivashankar, Alford, and Aha 2017), which uses landmarks to compute a heuristic for the hierarchical planning problem. On the code level, PTN-Plan has an interface for the node evaluation function, which can be used to setup a problem-specific f-function. In principle, this f could also be additive, so the idea of PANDA could be used in PTN-Plan in principle if the actual cost measure is additive.

Experimental Evaluation

To assess the role of the different extensions to the overall performance, we compare not only HTN with PTN-Plan

Dataset	#	PTN	HTN + IP	HTN + C	HTN	PTN	HTN + IP	HTN +C	HTN
car	4	0,72	3,11 ●	4,47 ●	4,38 ●	22,00	75,78 ●	113,20 ●	98,94 ●
page-blocks	5	5,50	23,00 ●	39,50 ●	30,67 ●	52,00	326,50 ●	743,00 ●	522,00 ●
analcat	6	2,32	9,94 ●	22,89 ●	22,47 ●	68,91	307,29 ●	561,33 ●	539,42 ●
segment	7	6,74	79,84 ●	46,12 ●	47,47 ●	89,52	1.327,05 ●	891,29 ●	891,29 ●
zoo	7	1,79	1,65	3,22 ●	2,92 ●	72,07	146,35 ●	106,67 ●	87,38 ●
autoUni	8	6,26	69,25 ●	28,21 ●	26,73 ●	91,83	817,60 ●	205,53 ●	194,27 ●
cnae9	9	277,18	-	-	-	111,82	-	-	-
mfeat-fourier	10	25,24	-	117,18 ●	123,87 ●	121,56	-	335,91 ●	338,43 ●
optdigits	10	36,82	-	167,68 ●	165,85 ●	95,18	-	173,32 ●	178,25 ●
pendigits	10	16,33	-	93,92 ●	92,88 ●	102,92	-	346,92 ●	347,33 ●
yeast	10	8,83	137,50 ●	21,00 ●	20,67 ●	153,67	1.361,50 ●	295,80 ●	265,00 ●
vowel	11	3,44	-	31,50 ●	22,14 ●	111,11	-	994,83 ●	361,71 ●
audiology	24	11,78	-	118,69 ●	116,21 ●	235,50	-	849,85 ●	740,05 ●
letter	26	41,32	-	-	-	219,40	-	-	-
kropt	28	141,55	-	-	-	405,00	-	-	-

Table 1: Comparison of PTN-Plan with other extensions of HTN planning. The column entitled with # shows the number of classes for the respective dataset. The left main column reports the average runtime to the first solution in seconds. The second main column reports the number of nodes generated. A hyphen means that no solution was found in the timeout.

but also with other variants. More precisely, we consider the version of HTN but with constant creation or efficient creation simulation (HTN + CC) and the version of HTN with both constant creation and interpreted predicates (HTN + IP). That is, PTN and HTN + IP use the above formalization that adopts interpreted predicates; PTN outsources the *refine* task to an oracle (described below). HTN and HTN + CC use the initial formalization without interpreted predicates. In the case of plain HTN, the generation of objects is simulated naively and *without* the encoding shown in the first formalization; a formalization like this was used in (Klusch, Gerber, and Schmidt 2005).

The BF-ε search is informed by a simple f-function that completes the partial dichotomies using a technique called RPND (Leathart, Pfahringer, and Frank 2016) and then computes the performance of that dichotomy. Note that this f is not optimistic but rarely overestimates the optimal cost by large margin. The ε is considered as an absolute value (instead of a relative one) of 1% accuracy tolerance. The same technique is used by the oracles to generate a moderate number of possible refinements.

Our evaluation is based on a couple of datasets of different numbers of classes. This is because the search graph structure and size highly depends on the number of classes. In the general HTN encoding, the search graph size grows in a factorial order with the number of classes. Note that even if we use oracles and their massive pruning, the search space still grows quite rapidly simply because more decisions must be made in total; in fact, even the run time of a hill climber increases as least linearly. The datasets are from a well-known repository called UCI (Asuncion and Newman 2007) and are used frequently to evaluate the performance of algorithms that create nested dichotomies (Leathart, Pfahringer, and Frank 2016). All the datasets are available at `http://openml.org`.

Planning for AutoML pipelines is afflicted much more by random effects than planning in other domains due to intrinsic randomized aspects. The main sources of randomness are the *splits* made on the given data set. That is, to check the quality of a solution, the data set is initially split into two parts, the so called training set, which is used to guide the search, and test set, which is used to evaluate the quality of a solution (by comparing the dichotomy's prediction for each of these instances with the true class). In our experiments, this split is always 70%/30%. The choice of this split has paramount effects on the evaluation of the candidates, so it is necessary to consider not only one such split but several ones in order to get a more stable estimate. Also, the evaluation during search makes such splits which is why the evaluation of nodes is always subject to a certain degree of randomness. Since the search itself is not aware of the random nature of these values, it is important to obtain a relatively stable estimate for the mean, which is then the ultimate goal of optimization.

As a consequence, we report the mean values over 25 experiments for each data set. Table 1 shows the results of our computations. The computations were executed on 16 Linux machines in parallel, each of which with a resource limitation of 16 cores (Intel Xeon E5-2670, 2.6Ghz) and 16GB memory. Each experiment was conducted on a associated with a timeout of 5 minutes. We do not report the solution quality since this is not an important measure for the comparison of the search space exploration. However, we briefly summarize that PTN-Plan never produced significantly worse solutions than any of the other algorithms.

PTN-Plan clearly dominates each of the other algorithm variants in both runtime and node generation. Significant improvements (5%-threshold in t-test) are indicated by ●. For three of the problems, it is the only algorithm that identifies the solution within the given time bound. PTN-Plan generates only half the number of nodes as standard HTN on all dataset and sometimes 10 times less (page-blocks).

The improvement of PTN-Plan over HTN + IP motivates the use of oracle tasks in this context. Interpreted predicates allow for a very simple problem specification but yield an exponential number of successors for the refinement nodes, which frequently produces memory overflows. In PTN, only a small subset of those nodes is actually created, which makes it much more scalable.

These results, though rather preliminary, strongly motivate the usage of oracles in hierarchical planning. It is not at all clear that general purpose heuristics, despite all their advantages, can achieve the same performance as obtained using oracles (with very domain-specific heuristic knowledge). For the time-being, no such heuristics are in sight.

Conclusion

We have introduced an extension to classical HTN planning called PTN (Programmatic Task Networks) that connects the planner with external libraries in the form of logic theories and oracles. The theories are used to evaluate function terms and predicates that may occur in the preconditions of operations and methods. Oracles are used by the planner to outsource the generation of sub-solutions for specific tasks. We have conducted an experimental evaluation in the area of automated machine learning (AutoML), which was also our motivation to use (and extend) hierarchical planning. While the concrete problem of creating a nested dichotomy can also be solved easily without planning, HTN is a great framework to describe the construction mechanism of more general machine learning pipelines; PTN of HTN paves the way for a more efficient construction of those pipelines.

Open issues are on both theoretical and practical sides. Theoretically, it would be interesting to learn more about the possibility to transfer general heuristics from classical planning or HTN planning to PTN. On the practical side, PTN-Plan is still very preliminary and only supports totally ordered networks, so there is also a great deal of engineering research ahead.

Acknowledgements This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901).

References

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. IJCAI*, 3022–3029.

Asuncion, A., and Newman, D. 2007. UCI machine learning repository.

Bercher, P., et al. 2015. Hybrid planning theoretical foundations and practical applications.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2962–2970.

Frank, E., and Kramer, S. 2004. Ensembles of nested dichotomies for multi-class problems. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*.

Geffner, H. 2000. Functional strips: a more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence*. Springer. 187–209.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning - theory and practice*. Elsevier.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*.

Hoffmann, J.; Bertoli, P.; Helmert, M.; and Pistore, M. 2009. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research* 35:49–117.

Hoffmann, J. 2003. The metric-ff planning system: Translating“ignoring delete lists”to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Klusch, M.; Gerber, A.; and Schmidt, M. 2005. Semantic web service composition planning with owls-xplan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, 55–62.

Leathart, T.; Pfahringer, B.; and Frank, E. 2016. Building ensembles of adaptive nested dichotomies with random-pair selection. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 179–194.

Mohr, F. 2017. *Towards automated service composition under quality constraints*. Ph.D. Dissertation, Paderborn University.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *J. Artif. Intell. Res. (JAIR)* 20:379–404.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *AAAI*, 3658–3664.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. AAMAS*, 981–988.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. S. 2013. The godel planning system: A more perfect union of domain-independent and hierarchical planning. In *IJCAI*, 2380–2386.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The anml language. In *Proc. KEPS*.

Thornton, C.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA*, 847–855.

Weber, I. M. 2009. *Semantic Methods for Execution-level Business Process Modeling: Modeling Support Through Process Verification and Service Composition*. Springer.

Tracking Branches in Trees – A Propositional Encoding for Solving Partially-Ordered HTN Planning Problems

Gregor Behnke and **Daniel Höller** and **Susanne Biundo**
Institute of Artificial Intelligence, Ulm University, D-89069 Ulm, Germany
{gregor.behnke, daniel.hoeller, susanne.biundo}@uni-ulm.de

Abstract

Planning via SAT has proven to be an efficient and versatile planning technique. Its declarative nature allows for an easy integration of additional constraints and can harness the progress made in the SAT community without the need to adapt the planner. However, there has been only little attention to SAT planning for hierarchical domains. To ease encoding, existing approaches for HTN planning require additional assumptions, like non-recursiveness or totally-ordered methods. Both limit the expressiveness of HTN planning severely. We propose the first propositional encodings which are able to solve general, i.e., partially-ordered, HTN planning problems, based on a previous encoding for totally-ordered problems. The empirical evaluation of our encoding shows that it outperforms existing HTN planners significantly.

Introduction

Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1996) is a versatile planning formalism, which has been used in many practical applications (Nau et al. 2005; Straatman et al. 2013; Champandard, Verweij, and Straatman 2009; Dvorak et al. 2014). It extends classical planning by introducing abstract tasks in addition to primitive (classical) actions. They represent portfolios of more complex courses of action which – if executed – achieve the abstract task. Decomposition methods map abstract tasks to partially-ordered sets of other tasks (that might be primitive or abstract) – and by that express the connection between higher- and lower-levels of action abstraction. Decomposition is continued until all tasks are primitive and these actions can be executed in the initial state. This decompositional structure is a powerful way to describe the set of possible solutions, making HTN planning more expressive than classical planning (Erol, Hendler, and Nau 1996; Höller et al. 2014; Höller et al. 2016). To solve HTN planning problems, fast and domain-independent planning systems are required that are informed about both – hierarchy and state. But as of now, the research in this area lacks behind that in classical planning. Most current HTN planners are based on heuristic search, as in classical planning. In classical planning, SAT-based planning has also proven to be highly efficient and has advantages compared to planning via heuristic search. Most notably, SAT-based planners benefit from future progress in SAT research without the need to adapt the planner – simply replacing the solver is sufficient. Also propositional encodings are easily extendable, e.g., to add further constraints, like goals formulated in LTL. Lastly propositional logic seems to be a suitable means to solve HTN planning problems, as verifying solutions was shown to be $\mathbb{NP}$-complete (Behnke, Höller, and Biundo 2015).

In HTN planning, there has been little research on SAT-based techniques. Most importantly, there is no SAT-based HTN planner capable of handling all HTN planning problems. There are only two restricted encodings, one by Mali and Kambhampati (1998) – which (among other restrictions) cannot handle recursion, and one by Behnke, Höller, and Biundo (2018) – which cannot handle partial order in methods, but can handle recursion. Both restrictions limit the expressiveness of HTN planning severely (Höller et al. 2014; Erol, Hendler, and Nau 1996) and limit the domain-modeller's freedom unnecessarily. We present the first encoding that can handle all propositional HTN planning problems.

We will show how the encoding of Behnke, Höller, and Biundo (2018) can be adapted such that it can also be applied to partially ordered domains. Since in that case, any ordering information in the encoding is lost, we propose a mechanism for representing the ordering constraints contained in the domain by additional decision variables. Since the order between two primitive tasks can only originate from a single method, this encoding is fairly compact.

Our empirical evaluation compares our encoding against state-of-the-art HTN planners. Here, we have considered combinatorial HTN planning problems, and not those where the HTN is hand-coded to help the planner find a solution. Our SAT-planner outperforms existing HTN planning techniques on these domains, some of them significantly.

First we introduce HTN planning formally and discuss related work. Then, we review the concept of totally-ordered Path Decomposition Trees and the SAT formula based on them. In section five, we introduce the concept of partially-ordered Path Decomposition Trees and present our SAT formula that can be used for planning in such domains. In the following chapter we describe the evaluation we conducted.

Preliminaries

We use the HTN formalism of Geier and Bercher (2011), where plans (partially ordered sets of task) sare represented by task networks.

Definition 1 (Task Network). *A task network tn over a set of task names X is a tuple $(T, \prec, \alpha)$, where*

- *T is a finite, possibly empty, set of tasks*
- *$\prec \subseteq T \times T$ is a strict partial order on T*
- *$\alpha : T \to X$ labels every task with a task name*

TN_X denotes the set of all task networks over the task names X. We write $T(tn) = T$, $\prec(tn) = \prec$ and $\alpha(tn) = \alpha$ for a task network $tn = (T, \prec, \alpha)$. Two task networks $tn = (T, \prec, \alpha)$ and $tn' = (T', \prec', \alpha')$ are *isomorphic*, written $tn \cong tn'$, iff a bijection $\sigma : T \to T'$ exists, s.t. $\forall t, t' \in T$ it holds that $(t, t') \in \prec$ iff $(\sigma(t), \sigma(t')) \in \prec'$ and $\alpha(t) = \alpha'(\sigma(t))$. Next we define the restriction notation.

Definition 2 (Restriction). *Let $R \subseteq D \times D$ be a relation, $f : D \to V$ a function and tn be a task network. Then:*

$$R|_X = R \cap (X \times X) \qquad f|_X = f \cap (X \times V)$$
$$tn|_X = (T(tn) \cap X, \prec(tn)|_X, \alpha(tn)|_X)$$

An HTN planning problem is defined as follows.

Definition 3 (Planning Problem). *A* planning problem *is a 6-tuple $\mathcal{P} = (L, C, O, \gamma, M, c_I, s_I)$, with*

- *L, a finite set of* proposition symbols
- *C, a finite set of* compound task names
- *O, a finite set of* primitive task names *with $C \cap O = \emptyset$*
- *$\gamma : O \to 2^L \times 2^L \times 2^L$, defining the preconditions and effects of each primitive task*
- *$M \subseteq C \times TN_{C \cup O}$, a finite set of* decomposition methods
- *$c_I \in C$, the* initial task name
- *$s_I \in 2^L$, the* initial state

The state transition semantics of primitive task names $o \in O$ is that of classical planning, given in terms of an precondition-, an add-, and a delete-list: $\gamma(o) = (prec(o), add(o), del(o))$. A primitive task is applicable in a state $s \subseteq L$ iff $prec(o) \subseteq s$ and its application results in the state $\delta(s, o) = (s \setminus del(o)) \cup add(o)$. A sequence of primitive tasks $o_1, \ldots, o_m$ is applicable in a state s_0 iff there exist states $s_1, \ldots, s_n$, each o_i is applicable in s_{i-1}, and $\delta(s_{i-1}, o_i) = s_i$. We define $M(c) = \{(c, tn) \mid (c, tn) \in M\}$ to be the methods applicable to c.

To obtain a solution in HTN planning, one starts with the initial compound task and repeatedly applies *decomposition methods* to compound tasks until all tasks in the current task network are primitive.

Definition 4 (Decomposition). *A method $m = (c, tn_m) \in M$ decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into a task network tn_2 by replacing the task t, written $tn_1 \xrightarrow[t,m]{} tn_2$, if and only if $t \in T_1$, $\alpha_1(t) = c$, and $\exists tn' = (T', \prec', \alpha')$ with $tn' \cong tn_m$ and $T' \cap T_1 = \emptyset$, where*

$$\begin{aligned} tn_2 &= (T'', \prec_1 \cup \prec' \cup \prec_X, \alpha_1 \cup \alpha')|_{T''} \text{ with} \\ T'' &= (T_1 \setminus \{t\}) \cup T' \\ \prec_X &= \{(t_1, t_2) \in T_1 \times T' \text{ with } (t_1, t) \in \prec_1\} \cup \\ &\quad \{(t_1, t_2) \in T' \times T_1 \text{ with } (t, t_2) \in \prec_1\} \end{aligned}$$

We write $tn_1 \to_D^* tn_2$, if tn_1 can be decomposed into tn_2 using an arbitrary number of decompositions.

Using the previous definition we can describe the set of solutions to a planning problem $\mathcal{P}$.

Definition 5 (Solution). *A task network tn_S is a solution to a planning problem $\mathcal{P}$, if and only if*

(1) there is a linearisation $t_1, \ldots, t_n$ of $T(tn_S)$ according to $\prec(tn_S)$,

(2) $\alpha(tn_S)(t_1), \ldots, \alpha(tn_S)(t_n)$ is executable in s_I, and

(3) $(\{1\}, \emptyset, \{(1, c_I)\}) \to_D^* tn_S$,

$\mathfrak{S}(\mathcal{P})$ denotes the sets of all solutions of $\mathcal{P}$, respectively.

Note that this definition of HTN planning problems excludes some of the features in the original formulation by Erol, Hendler, and Nau 1996. His formalisation allows for constraints to be present in task network, namely before, after, and between constraints. The constraint type used most often, are before constraints, which correspond to SHOP(2)'s method preconditions. Our planner can handle them by compiling them into additional actions, as does SHOP2. So far, we don't support other constraint types.

To show that a task sequence π is a solution to a planning problem, we use *Decomposition Trees* (DTs) as witnesses (Geier and Bercher 2011). They describe how π can be obtained from the initial abstract task via decomposition.

Definition 6. *Let $\mathcal{P} = (L, C, O, M, c_I, s_I)$ be an HTN planning problem. A valid decomposition tree T is a 5-tuple $T = (V, E, \prec, \alpha, \beta)$, where*

1. *(V, E) is a directed tree with a root-node r.*
2. *$\prec \subseteq V \times V$ is a strict partial order on V and is inherited along the tree, i.e., if $a \prec b$, then $a' \prec b$ and $a \prec b'$ for any children a' of a and b' of b.*
3. *$\alpha : V \to C \cup O$ assigns each inner node an abstract task and each leaf a primitive task.*
4. *$\beta : V \to M$ assigns each inner node a method.*
5. *$\alpha(r) = c_I$*
6. *for all inner nodes $v \in V$ with $\beta(v) = (c, tn)$ and children $ch(v) = \{c_1, \ldots, c_n\}$, it holds that $c = \alpha(v)$. Further, a bijection $\phi : ch(v) \to T(tn)$ must exist with $\alpha(c_i) = \alpha(tn)(\phi(c_i))$ for all c_i, and $c_i \prec c_j$ iff $\phi(c_i) \prec(tn) \phi(c_j)$.*

$\prec$ may not contain orderings apart those induced by 2. or 6. The yield $yield(T)$ of T is the task network induced by the leafs of T, i.e. V, α, and $\prec$ restricted to these leafs.

Geier and Bercher (2011) showed the following theorem:

Theorem 1. *Given a planning problem $\mathcal{P}$, then for every task sequence π the following holds:*
There exists a valid decomposition tree T s.t. π is a linearisation of $yield(T)$ if and only if $\pi \in \mathfrak{S}(\mathcal{P})$.

This means, that instead of finding a solution to the planning problem $\mathcal{P}$, we can equivalently try to find a DT whose yield is executable – the approach we use in this paper.

Related Work

Past research has already investigated possible translations of HTN planning problems into logic.

HTNs and Logic

Notably, Mali and Kambhampati (1998) proposed a SAT-translation for HTNs. Their HTN formalism differs significantly from the established HTN formalism, making their encoding simpler and different from ours. They allow inserting tasks into task networks apart from decomposition and do not specify an initial task. Furthermore their encoding is also restricted to non-recursive domains. Such domains can be translated into an equivalent STRIPS planning problem, which is not the case for general domains (Höller et al. 2014). Dix, Kuter, and Nau (2003) have proposed an encoding of totally-ordered HTN planning into answer set programming, mimicking the search of SHOP. Their evaluation shows that the translated domain performs significantly worse than the SHOP algorithm (up to a factor of 1.000).

PDT-based encoding

Since our work is based on the encoding presented by Behnke, Höller, and Biundo (2018), we start by reviewing this encoding in detail. Their idea was to restrict the maximum depth of decomposition. The planner start with some small bound K and constructs a SAT formula satisfiable if a solution with depth $\leq K$ exists. If not, K is increased and the process is repeated. To construct this formula, they used a compact representation of all possible decompositions with depth $\leq K$ – the Path Decomposition Tree PDT P. A satisfying valuation of the SAT formula then represents a decomposition tree T that is a subgraph of P. They however studied PDTs and the resulting formula only in the context of totally-ordered HTN planning, which is as we have argued in the introduction far less expressive and versatile than full partially-ordered HTN planning. Also we want to note, that almost all current HTN planning systems are constructed for partially-ordered domains, as most domains used in practice are partially ordered.

A PDT is a compact representation of all possible decompositions of the initial abstract task up to a given depth-bound K. Every such decomposition is represented by a decomposition tree (see Def. 6). The PDT is then a graph P such that it contains every possible decomposition tree as one of its subgraphs P'. To ensure a "common structure" we also require that the root of P' is the root of P. Next we give the formal definition of totally-ordered Path Decomposition Trees. To ease notation, we denote with $\mathfrak{L}(T = (V, W))$ the set of all leafs of a tree T.

Definition 7. *Let $\mathcal{P} = (L, C, O, M, c_I, s_I)$ be a planning problem and K a height bound. A Path Decomposition Tree P_K of height K is a triple $P_K = (V, E, \alpha)$ where*

1. *V are the nodes of a tree of height $\leq K$, with edges given by function $E : V \to V^*$, and which has the root node r.*
2. *$\alpha : V \to 2^{C \cup O}$ assigns each node a set of possible tasks.*
3. *$c_I \in \alpha(r)$*
4. *for all inner nodes $v \in V$, for each abstract task $c \in \alpha(v) \cap C$ that can be assigned to that node, and for each method $(c, tn) \in M(c)$, there exists a subsequence $v_1, \ldots, v_{|T(tn)|}$ of the children $E(v)$, such that $tn_i \in \alpha(v_i)$ for all $i \in \{1, \ldots, |T(tn)|\}$, where tn_i is the i^{th} element of the sequence of task names of tn.*
5. *$\forall v \in \mathfrak{L}(V, E)$: either $\alpha(v) \subseteq O$ or the height of v is K.*

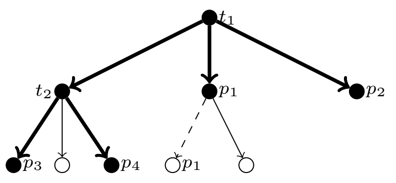

Figure 1: An example PDT, a DT as its subgraph (nodes filled), and the extension for primitive tasks (dashed line). The nodes of the DT are each annotated with the task (t_i for abstract and p_i for primitives ones) that they are be labelled with in the DT. The node labelled p_2 does not have children even though it is not at the "lowest" level due to the fact that it can only be labelled with primitive tasks (p_2 in our example), while the node labelled with p_1 can potentially also be labelled with an abstract task. For this consider e.g. the methods $t_1 \mapsto t_2, p_1, p_2$ and $t_1 \mapsto t_2, t_3, p_2$. Note that there is one non-filled node that is also labelled with a task. This is an encoding trick to ensure that the leafs of the DT are also leafs of the PDT – primitive tasks are simply "inherited" by one of their children in the PDT.

This definition assumes that the tasks in a method's task network are totally-ordered and thus can be projected directly to a totally-ordered sequence of children. As a result, the leafs of the PDT are also totally-ordered (according to the order implied by their common ancestors). Behnke, Höller, and Biundo (2018) provide an algorithm constructing a PDT P_K^σ given a so-called child-arrangement function σ. Based on it, they describe a SAT-formula $\mathcal{F}_D(\mathcal{P}, K)$ that is satisfiable if and only if there exists a subgraph G' of the PDT P_K^σ that forms a valid decomposition tree. A satisficing valuation of $\mathcal{F}_D(\mathcal{P}, K)$ represents such a DT G' – expressed by two types decision variables:

- t^v – v is part of G' and is labelled with t, i.e., $\alpha(v) = t$.
- m^v – the method m was applied to the node v of G', i.e., $\beta(v) = m$

Their encoding propagates primitive tasks occurring at any node v downwards through the first child of v in the PDT. This ensured that $yield(G')$ is represented by the leafs of P_K^σ that have a task assigned to them – else inner nodes of P_K^σ may belong to the yield. In addition to $\mathcal{F}_D(\mathcal{P}, K)$, Behnke, Höller, and Biundo used a second formula $\mathcal{F}_E(\mathcal{P}, K)$ ensuring executability of the tasks assigned to the leafs of G'.

For the formula $\mathcal{F}_D(\mathcal{P}, K)$ – and for other formulae thereafter, we use the functor $\mathbb{M}(V)$, which given a set of decision variables V, outputs a formula that is satisfiable if and only if at most one of them (Sinz 2005). $\mathcal{F}_D(\mathcal{P}, K)$ consist solely of local constraint, i.e., one sub-formula is generated per node of the PDT. The formula to be generated for a node v of the PDT $P_K^\sigma = (V, E, \alpha)$ is either $\mathbb{M}(\{t^v \mid t \in \alpha(v) \cap O\}) \wedge_{c \in C} \neg c^v$ if $v \in \mathcal{L}(P_K^\sigma)$, i.e., if v is a leaf, or else the following formula:

$$f(v) = \mathbb{M}(\{t^v \mid t \in \alpha(v)\}) \wedge \mathit{selectMethod}(v) \\ \wedge \mathit{applyMethod}(v) \wedge \mathit{inheritPrimitive}(v) \\ \wedge \mathit{nonePresent}(v)$$

It first asserts that every node in the decomposition tree can be labelled with at most one task. The next four sub-formulae encode the further restrictions a decomposition tree must fulfil. *selectMethod* ensures that an applicable method is chosen and that only one is chosen, provided v is labelled with an abstract task.

$$\mathit{selectedMethod}(v) = \mathbb{M}(\{m^v \mid M(\alpha(v) \cap C)\}) \wedge \\ \left[\bigwedge_{t\in\alpha(v)\cap C}\left(t^v \rightarrow \bigvee_{m\in M(t)} m^v\right)\right] \wedge \left[\bigwedge_{m\in M(\alpha(t)\cap C)} (m^v \rightarrow t^v)\right]$$

applyMethod forces that whenever a method is selected, the tasks in its task network are assigned to the children of v. Let for a method $m = (c, tn)$ be $v_1, \ldots, v_{|T(tn)|}$ the subsequence given in Def. 6. Let further denote $t_{tn,i}$ the ith task of the (totally-ordered) task network tn.

$$\mathit{applyMethod}(v) = \bigwedge_{m=(t,tn)\in M(\alpha(v))} \Bigg[m^v \rightarrow \\ \left(\bigwedge_{i=1}^{|tn|} t_{tn,i}^{v_i} \wedge \bigwedge_{v_i\in E(v)\setminus\{v_1,\ldots,v_{|tn|}\}} \bigwedge_{t_*\in C\cup O} \neg t_*{}^{v_i}\right)\Bigg]$$

These clauses also propagate the total order between the subtasks $v_1, \ldots, v_{|tn|}$. *inheritPrimitive* and *nonePresent* take care of the border cases, where v is either assigned a primitive task, or none at all. Let here be v_1 the first node in $E(v)$.

$$\mathit{inheritPrimitive}(v) = \\ \bigwedge_{p\in\alpha(v)\cap O}\left[p^v \rightarrow \left(p^{v_1} \wedge \bigwedge_{v_i\in E(v)\setminus\{v_1\}} \bigwedge_{k\in C\cup O} \neg k^{v_i}\right)\right]$$

$$\mathit{nonePresent}(v) = \left(\bigwedge_{t\in\alpha(v)} \neg t^v\right) \rightarrow \left(\bigwedge_{v_i\in E(v)} \bigwedge_{t\in C\cup O} \neg t^{v_i}\right)$$

The full decomposition formula $\mathcal{F}_D(\mathcal{P})$ is then simply $\bigwedge_{v\in V} f(v)$.

Partially-Ordered Decomposition

We can extend this encoding, allowing us to track the partial order induced by the methods. As a first step, we have to ignore the fact that the PDT represents any ordering constraint. For that purpose, we introduce unordered PDTs, which differ only slightly from PDTs. Unordered PDTs – as their names suggests – don't have an ordering on the children of a node. Based on this, the main difference lies in 4. of the definition. For PDTs every node and applicable method, the subtasks of that method must from a subsequence of the nodes children, while for an unordered PDT it suffices that they are a subset.

Definition 8. *Let $\mathcal{P} = (L, C, O, M, c_I, s_I)$ be a planning problem and K a height bound. An unordered PDT P_K of height K is a triple $P_K = (V, E, \alpha)$ where*

1. *(V, E) is a tree of height $\leq K$ with the root node r.*
2. *$\alpha : V \rightarrow 2^{C\cup O}$ assigns each node a set of possible tasks.*
3. *$c_I \in \alpha(r)$*
4. *for all inner nodes $v \in V$, for each abstract task $c \in \alpha(v) \cap C$ that can be assigned to v, and for each method $(c, tn) \in M(c)$, there exists a subset $D = \{v_1, \ldots, v_{|T(tn)|}\}$ of v's children, such that a bijection $\phi^v_{(c,tn)} : D \rightarrow T(tn)$ exists with $\alpha(tn)(\phi^v_{(c,tn)}(d)) \in \alpha(d)$ for all $d \in D$*
5. *$\forall v \in \mathfrak{L}(V, E)$: either $\alpha(v) \subseteq O$ or the height of v is K.*

As uPDTs are a structural relaxation of PDTs, we can use the same generation procedure based on a child-arrangement function σ – simply by ignoring that methods are partially ordered – we use some topological ordering of the methods for generating P^σ_K instead. Based on the generated uPDT, we can also use the same formula $\mathcal{F}_D(\mathcal{P}, K)$ describing decomposition. To capture the partial order we add new decision variables for bookkeeping:

- b^v_w – for nodes v and w that have the same parent, i.e., are siblings. If b^v_w is true, the order $v \prec w$ is contained in the method applied to the parent of v and w.

These variables are sufficient to infer the order between all elements of $\mathit{yield}(G')$. This is due to how order is inherited in a decomposition tree. Essentially, the order between two nodes v and v' can only stem from the method applied to their last common ancestor in G'. The structure is illustrated in Figure 2. For two leafs v and v' of the tree, let $\mathfrak{A}(v, v')$ be the last common ancestor of v and v'. Further be $\mathfrak{C}(a, v)$, be the child c of a, s.t. the leaf v is below c. Then v stems from $\mathfrak{C}(\mathfrak{A}(v, v'), v)$, while v' from $\mathfrak{C}(\mathfrak{A}(v, v'), v')$. Then the formal property is the following:

Theorem 2. *Let $T = (V, E, \prec, \alpha, \beta)$ be a decomposition tree. Let $v, v' \in \mathfrak{L}(V, E)$ be two leafs of T, $c = \mathfrak{A}(v, v')$ be the last common ancestor of v and v'. Then the order between v and v' is the same as between $v_c = \mathfrak{C}(c, v)$ and $v'_c = \mathfrak{C}(c, v')$ induced by the method applied to c.*

Proof. Suppose there is an order between v_c and v'_c. Then by 2. of Def. 6, this order must also be present between v and v'.
Suppose there is no order between v_c and v'_c. Then the direct children of v_c and v'_c that are ancestors of v and v' respectively cannot contain any order, too. By definition, any order between them must either be introduced by methods or by 2. of Def. 6. Clearly, no decomposition methods could have introduced the ordering since the tasks don't have a common parent. Also since v_c and v'_c have no order between them 2. of Def. 6 is not applicable. By induction, we can conclude that there is not order between v and v'. □

To keep track of the ordering constraints, we have to add for every decision variable m^v clauses that enforce that the correct b^v_w variables are set true. We therefore add for every m^v the following clauses to $F_D(\mathcal{P}, K)$, where $m = (c, tn)$,

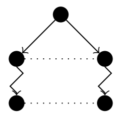

Figure 2: An illustration where order originates from in a decomposition tree.

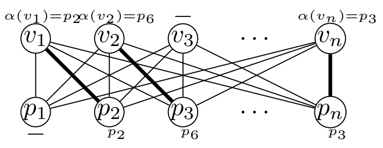

Figure 3: Matching structure between leafs of P_K^σ, and positions in the primitive sequence.

$\{v_1, \dots, v_n\}$ are the nodes of P_K^σ to which the tasks of tn are mapped, and $\{t_1, \dots, t_n\}$ be those tasks.

$$\bigwedge_{i=1}^{n} \bigwedge_{j \in \{1,\dots,n\} \text{ s.t. } (t_i,t_j) \in \prec(tn)} (m^v \rightarrow b_{v_j}^{v_i})$$

These clauses enforce that the b_w^vs represent a superset of the ordering constraints induced by the applied methods.

To complete the encoding we need a formula $\mathcal{F}_E(\mathcal{P}, K)$ that is satisfiable if and only if $yield(G')$ is executable. Let $l = |\mathfrak{L}(P_K^\sigma)|$ be the number of leafs of P_K^σ. We separate this formula into two parts: representing a linearisation of $yield(G')$ and checking that this linearisation is executable. A linearisation of $yield(G')$ is a mapping of the leafs of G' to a sequence of *positions*. We can use l as an upper bound to the number of positions – and we have always used this value in our encoding. Also we denote these positions as $1, \dots, l$. This mapping is essentially a bipartite matching that must not contradict the ordering constraints. Figure 3 illustrates these structures.

We have to generate a SAT formula that represents such a matching and is only satisfiable iff the matching is valid (i.e. an actual matching and it respects the order). We omit a formal proof of correctness, as we deem the encoding straightforward enough to be considered correct by construction. We introduce two new decision variables:

- c_i^v – leaf v connected with position i
- a^v – leaf v contains a task (i.e. is a leaf of G' and has to be matched)

Based on these variables, we can formulate the restrictions a valid matching must fulfil. First, every leaf or position may be matched only once.

$$F_1 = \bigwedge_{i=1}^{l} \mathbb{M}(\{c_i^v \mid v \in \mathfrak{L}(P_K^\sigma)\}) \wedge \bigwedge_{v \in \mathfrak{L}(P_K^\sigma)} \mathbb{M}(\{c_i^v \mid 1 \leq i \leq l\})$$

Next, we define the a^v atoms, that are true exactly if the leaf v of P_K^σ contains an action. We use them as intermediate variables to decrease the overall size of the formula.

$$F_2 = \bigwedge_{v \in \mathfrak{L}(P_K^\sigma)} \left[\left(\neg a^v \rightarrow \bigwedge_{o \in \alpha(v)} \neg o^v \right) \wedge \left(a^v \rightarrow \bigvee_{o \in \alpha(v)} o^v \right) \right]$$

Next, a leaf of P_K^σ that contains a task has to be matched – else it would be allowed to disregard it when checking the executability of $yield(G')$.

$$F_3 = \bigwedge_{v \in \mathfrak{L}(P_K^\sigma)} \left[\left(\neg a^v \rightarrow \bigwedge_{1 \leq i \leq l} \neg c_i^v \right) \wedge \left(a^v \rightarrow \bigvee_{1 \leq i \leq l} c_i^v \right) \right]$$

If all these formulae are fulfilled, the atoms c_i^v represent a matching between all leafs of G' and the positions. As a next step, we have to ensure that this matching does not violate any ordering constraint induced by the chosen decomposition methods. To do that, we have to exclude the possibility that there are two positions $i < i'$ where the tasks they are matched with must occur in the opposite order. F_4 forbids the mentioned situation.

$$F_4 = \bigwedge_{i=1}^{l} \bigwedge_{i'=i+1}^{l} \bigwedge_{v,v' \in \mathfrak{L}(P_K^\sigma)} \left((c_i^v \wedge c_{i'}^{v'}) \rightarrow \neg b_{\mathfrak{C}(\mathfrak{A}(v,v'),v)}^{\mathfrak{C}(\mathfrak{A}(v,v'),v')} \right)$$

The second constraint states that the chosen linearisation of the tasks at the leafs of G' must be executable in the initial state. To express executability, we use the encoding proposed by Kautz and Selman (1996). For every proposition symbol $p \in L$, we introduce a decision variable p^i for $0 \leq i \leq L$. p^i is true if p is true after executing the ith action. Further, we introduce decision variables t^i for every primitive task $t \in O$, stating that t is executed at timestep i. Then the formula $\mathcal{F}_{LE}$ is defined as follows:

$$\mathcal{F}_{LE} = \bigwedge_{p \in s_I} p^0 \wedge \bigwedge_{p \in L \setminus s_I} \neg p^0 \wedge \bigwedge_{i=0}^{l-1} (\mathcal{F}_A(i) \wedge \mathcal{F}_M(i)) \wedge \bigwedge_{i=1}^{l} \mathbb{M}(\{t^i \mid t \in O\})$$

$$\mathcal{F}_A(i) = \bigwedge_{t \in O} t^{i+1} \rightarrow \left(\bigwedge_{p \in prec(t)} p^i \wedge \bigwedge_{p \in add(t)} p^{i+1} \wedge \bigwedge_{p \in del(t)} \neg p^{i+1} \right)$$

$$\mathcal{F}_M(i) = \bigwedge_{p \in L} \left[(\neg p^i \wedge p^{i+1}) \rightarrow \bigvee_{t \in O \text{ with } p \in add(t)} t^{i+1} \right]$$

So far, we have only checked that the matching is valid and that the sequence of actions assigned to the positions is executable, but not that the matching influences the tasks assigned to positions. I.e., we have to add two more formulae that express that if a position it not matched to any leaf, then it also cannot contain a task, and that if it is matched it has to contain exactly the same task as the leaf does.

$$F_5 = \bigwedge_{1\le i\le l} \left[\left(\bigwedge_{v\in\mathfrak{L}(P_K)} \neg c_i^v \right) \rightarrow \left(\bigwedge_{t\in O} \neg t^i \right) \right]$$

$$F_6 = \bigwedge_{v\in\mathfrak{L}(P_K)} \bigwedge_{t\in\alpha(v)} \bigwedge_{1\le i\le l} t^v \wedge c_i^v \rightarrow t^i$$

To sum up, the full formula expressing executability is:

$$\mathcal{F}_E(\mathcal{P}, K) = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6 \wedge \mathcal{F}_{LE}$$

We know that the satisfying valuations of $\mathcal{F}_D(\mathcal{P}, K)$ represent exactly all decomposition trees of $\mathcal{P}$ with an height $\le K$ (Behnke, Höller, and Biundo 2018). Based on this, the correctness and completeness of our encoding can be shown.

Theorem 3. *$\mathcal{F}_E(\mathcal{P}, K) \wedge \mathcal{F}_D(\mathcal{P}, K)$ is satisfiable iff $\mathcal{P}$ has a solution with decomposition height $\le K$.*

Proof. $\Rightarrow$: Let ν be a satisfying valuation of $\mathcal{F}_E(\mathcal{P}, K) \wedge \mathcal{F}_D(\mathcal{P}, K)$. Then ν represents a decomposition tree, since $\mathcal{F}_D(\mathcal{P}, K)$ is satisfied (Behnke, Höller, and Biundo 2018). Thus the tasks assigned to the leafs of the Path Decomposition Tree encoded by $\mathcal{F}_D(\mathcal{P}, K)$ from the yield Y of a Decomposition Tree. Also the sequence of actions S represented by the t^i is executable, due to $\mathcal{F}_{LE}$. What remains to show, is that this sequence is a linearisation of the yield Y. Due to $F_1 \wedge F_2 \wedge F_3$ the c_i^v represent a matching of Y to S and due to $F_5 \wedge F_6$ matched elements of Y and S contain the same task. Lastly, due to Theorem 2, the order between two tasks in Y depends solely on the method applied to their last common ancestor. Due to the clauses introducing the b_w^v variables, at least those orderings induced by the decomposition tree are true. Allowing for more order is not a problem, since ν already represents a linearisation. Lastly, F_4 ensures that the order encoded by the b_w^v is respected.

$\Leftarrow$: Let $T = (V, E, \prec, \alpha, \beta)$ be a decomposition tree whose yield is executable. Then a valuation ν exists that satisfies $\mathcal{F}_D(\mathcal{P}, K)$ (Behnke, Höller, and Biundo 2018) and represents T. Let $v_1, \dots, v_n$ be the leafs of the PDT who have a task assigned to them in ν. Let further be $i_1, \dots, i_n$ the indices of these tasks in the executable linearisation of the yield of T. We then set $c_{i_j}^{v_j}$ true for all $j \in \{1, \dots, n\}$. We also set the $\alpha(v_j)^{i_j}$ and the appropriate p^i true. Also we set b_w^v true as appropriate, which cannot violate the clauses of F_4, as the respective order must also be present in the yield of T. This valuation satisfies $\mathcal{F}_E(\mathcal{P}, K) \wedge \mathcal{F}_D(\mathcal{P}, K)$. $\square$

Evaluation

We have conducted an empirical evaluation of our planner to show that it performs favourably compared to other HTN planning systems. The code of our planner is available at `www.uni-ulm.de/in/ki/panda/`. Since most planning problems are given lifted, we use a combination of the planning graph and task decomposition graphs (Bercher et al. 2017) to ground them.

Domains. Our benchmarking set is composed of the following domains (will be released upon acceptance):

Domain	$\lvert L\rvert$		$\lvert O\rvert$		$\lvert C\rvert$		$\lvert M\rvert$	
	min	max	min	max	min	max	min	max
PCP	6	9	8	14	4	46	10	34
ENTERTAINMENT	10	146	16	455	10	170	20	541
UM-TRANSLOG	9	25	7	22	2	27	2	28
SATELLITE	6	37	7	123	3	25	10	214
WOODWORKING	10	101	7	739	4	443	9	2002
SMARTPHONE	10	103	8	231	3	66	4	360
ROVER	21	511	73	4257	14	285	49	3279
TRANSPORT	11	364	13	1968	11	802	21	3158

Domain	$\lvert\mathcal{L}(P_K)\rvert$		K		#clause		#plansteps	
	min	max	min	max	min	max	min	max
PCP	12	70	4	9	14.012	12.091.312	10	42
ENTERTAINMENT	8	78	4	6	416	42.028	7	42
UM-TRANSLOG	7	40	3	4	218	281.642	7	26
SATELLITE	5	40	3	5	183	1.375.308	5	20
WOODWORKING	3	25	3	7	531	689.552	3	19
SMARTPHONE	7	78	3	5	3.332	18.878.346	5	77
ROVER	53	61	5	5	4.048.432	7.045.922	27	36
TRANSPORT	8	48	4	6	3.980	5.176.067	8	42

- UM-TRANSLOG, WOODWORKING, SATELLITE, and SMARTPHONE are the benchmark domains of Bercher, Keen, and Biundo (2014).
- ENTERTAINMENT describes setting-up HiFi devices.
- ROVER is the domain used by Höller et al. (2018). It is based on the problem instances of the IPC3 domain ROVER combined with an HTN-structure similar to the one developed for SHOP.
- TRANSPORT describes a deliver-with-trucks scenario. There are several trucks (which do not need fuel) to deliver packages from their start location to a destination.
- PCP is an encoding of Post's Correspondence Problem. Since HTN planning is undecidable, we felt it proper to show that an HTN planner is able to solve undecidable problems (like PCP) when encoded in an HTN domain.

Behnke, Höller, and Biundo (2018) used the same domains except PCP in their evaluation. They, however, had to alter most of them, since these domains are naturally partially-ordered. In order for a totally-ordered HTN planner to be able to handle these benchmark domains, Behnke, Höller, and Biundo (2018) have manually added additional ordering constraints to each partially-ordered method. Adding ordering constraints to HTN domains can make them unsolvable (see e.g. PCP, which cannot contain a solution when totally ordered). The additional orderings were chosen such that at least one solution was retained. We also want to note that adding these orderings makes some of the domains much easier to solve. For example, in transport, interleaving using the partial order is required to find optimal solutions. If the domain is totally-ordered, one package has to be delivered before another package could be picked up. The domains ENTERTAINMENT, ROVER, and TRANSPORT contain method preconditions, which we compile away into additional actions preceeding all other actions.

Planners. Each planner was given 10 minutes runtime and 4 GB RAM per instance on an Intel Xeon E5-2660. We have compared all state-of-the-art HTN planning systems:

- SHOP2 (Nau et al. 2003) and PANDA's version of SHOP2,
- FAPE (Dvorak et al. 2014),
- UMCP (Erol, Hendler, and Nau 1994),

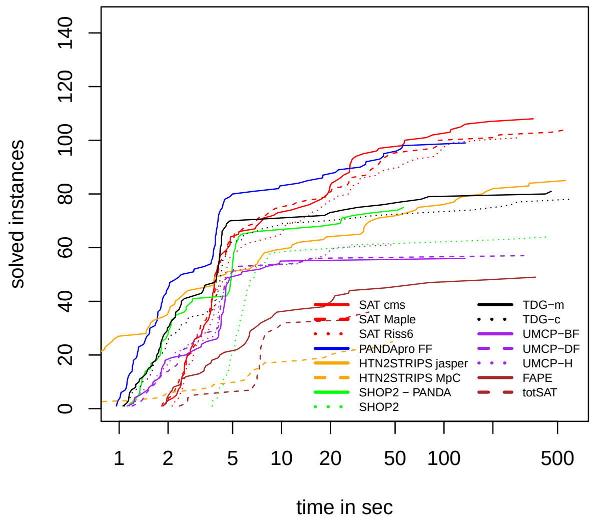

Figure 4: Runtime vs number of solved instances per planner

- PANDA with the TDG_m and TDG_c heuristics (Bercher et al. 2017) using greedy A*,
- PANDApro using the FF heuristic (Höller et al. 2018),
- HTN2STRIPS (Alford et al. 2016), and
- totSAT (Behnke, Höller, and Biundo 2018).

FAPE – according to the description in its paper – does not support recursive domains. Thus, we ran it only on the domains SATELLITE, WOODWORKING, and ROVER, which are the non-recursive ones in our evaluation. Similarly, as totSAT can only handle totally-ordered instance, we have run it only on those instances from our benchmark set that are totally ordered. Lastly, we have tested HTN2STRIPS with two different classical planners. We have used both jasper (which was originally used by Alford et al. (2016)) as well as Madagascar (Rintanen 2014), the currently best known SAT planner. We chose to do so, to compare our propositional encoding with the theoretically only so-far known propositional encoding for partially-ordered HTNs: first using the HTN2STRIPS translation and then the $\exists$-step encoding (Rintanen, Heljanko, and Niemelä 2006) for the resulting planning problem.

For our planner, we have evaluated three SAT solvers, each a top performers at the SAT Competition 2016. These were: cryptominisat5 (Soos 2016), MapleCOMSPS (Liang et al. 2016), and Riss6 (Manthey, Stephan, and Werner 2016). As our planner performs the translation using a bound K, we usually have to try several values for K. We started with $K = 1$ and increased by 1 if the formula was unsolvable. This iterative procedure allows us to handle any recursion in the domains, as we gradually unroll it.

Results. In Tab. 1 we show the number of solved instances per planner within the given time and memory limits. Fig. 4 shows the solved instances depending on runtime. First, our SAT-encoding, no matter the solver, solves more instances than any other planner. Second, our planner is on par in every domain with the best solver for that domain, or solves significantly more instances than other planners.

	#instances	SAT cms	SAT Maple	SAT Riss6	PANDApro FF	HTN2STRIPS jasper	HTN2STRIPS MpC	SHOP2 - PANDA	SHOP2	TDG-m	TDG-c	UMCP-BF	UMCP-DF	UMCP-H	FAPE	totSAT [AAAI18]
PCP	17	**11**	**11**	10	10	3	3	10	0	9	8	0	0	0	-	-
ENTERTAINMENT	12	**12**	**12**	**12**	11	5	4	9	5	9	9	5	5	6	-	12 / 12
UM-TRANSLOG	22	**22**	**22**	**22**	**22**	19	7	**22**	**22**	**22**	**22**	**22**	**22**	**22**	-	19 / 19
SATELLITE	25	**25**	24	23	**25**	23	8	19	22	**25**	21	18	20	23	22	5 / 5
WOODWORKING	11	**11**	**11**	**11**	10	5	4	6	8	8	10	6	6	6	0	-
SMARTPHONE	7	**7**	6	6	5	6	5	5	4	5	5	4	4	4	-	-
ROVER	20	4	4	4	3	**5**	4	3	3	2	2	0	0	0	3	-
TRANSPORT	30	16	14	13	13	**19**	3	1	0	1	1	1	0	0	-	-
total	144	**108**	104	101	99	85	38	75	64	81	78	56	57	61	25	36

Table 1: Number of solved instances per planner per domain. Maxima are indicated in bold. cms = cryptominisat5

We want to point out our performance in the domains TRANSPORT and PCP. In TRANSPORT we only solve 3 instances less than HTN2STRIPS, while all other planners solve at most a single instance. In PCP, we solve significantly more instances than HTN2STRIPS. This is notable, as both domains contain difficult combinatorially problem. This is especially notable, since HTN2STRIPS internally uses a state-of-the-art classical planner (jasper, (Xie, Müller, and Holte 2014)). However, there still seems to be room for improvement, as no planner seem to be well equipped to exploit the hierarchy in the ROVER domain.

The original totSAT for totally ordered domains has poor coverage, based on the fact that most domains of the benchmark set are partially-ordered. Lastly, we can observer that using Madagascar in conjunction with the HTN2STRIPS encoding seems to perform extremely poorly. In most instances Madagascar is aborted after only a few seconds as it reached the memory limit. This is probably due to the large number of groundings for the operators in the HTN2STRIPS encoding representing methods, which is a known problem of the encoding. We have re-run Madagascar with a memory limit of 20 GB instead of 4 GB and have only seen an increase by 4 solved instances. Also, the per-instance runtime when compared to jasper is fairly poor. We suppose that this is due to the way the encoding works. Modern SAT-based planning draws its efficiency mainly from the ability to execute several operators in parallel. This is not possible in the encoded domain as the next-predicates ensure that all simultaneously applicable actions form a clique in the disabling graph, i.e., cannot be executed parallel in the propositional encoding.

Conclusion

We have presented the first encoding for SAT-based HTN planning that can solve all propositional HTN planning problems. To that end, we have utilised a previous encoding that was only usable for totally-ordered planning, which restricts the freedom of the domain modeller unnecessarily, and extended it to partial order. Lastly, we have shown that our new planner outperforms state-of-the-art HTN planners. This planner has already been used in practice, namely in an assistant teaching users how to use electronic tools in Do-It-Yourself projects (Behnke et al. 2018).

Acknowledgments

This work was partly done within the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG).

References

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. of the 26th Int. Conf. on Autom. Plan. and Sched., (ICAPS 2016)*, 20–28. AAAI Press.

Behnke, G.; Schiller, M.; Kraus, M.; Bercher, P.; Schmautz, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2018. Instructing novice users on how to use tools in DIY projects. In *Proc. of the 27th Int. Joint Conf. on AI and the 23rd Europ. Conf. on AI (IJCAI-ECAI 2018).* AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proc. of the 25th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2015)*, 25–33. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-ordered hierarchical planning through SAT. In *Proc. of the 32th AAAI Conf. on AI (AAAI 2018)*, 6110–6118. AAAI Press.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. of the 26th Int. Joint Conf. on AI (IJCAI 2017)*, 480–488. AAAI Press.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of the 7th Ann. Symp. on Combinatorial Search (SoCS 2014)*, 35–43. AAAI Press.

Champandard, A.; Verweij, T.; and Straatman, R. 2009. The AI for Killzone 2's multiplayer bots. In *Proc. of the Game Developers Conference 2009 (GDC 2009).*

Dix, J.; Kuter, U.; and Nau, D. 2003. Planning in answer set programming using ordered task decomposition. In *Proc. of the 26th Annual German Conf. on AI (KI 2003)*, 490–504. Springer.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A flexible ANML actor and planner in robotics. In *Proc. of the 4th Work. on Plan. and Rob. (PlanRob 2014)*, 12–19.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. of the 2nd Int. Conf. on AI Plan. Systems (AIPS)*, 249–254. AAAI Press.

Erol, K.; Hendler, J.; and Nau, D. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI* 18(1):69–93.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the 22nd Int. Joint Conf. on AI (IJCAI 2011)*, 1955–1961. AAAI Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of the 21st Europ. Conf. on AI (ECAI 2014)*, volume 263, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of the 26th Int. Conf. on Autom. Plan. and Sched., (ICAPS 2016)*, 158–165. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, B. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proc. of the 28th Int. Conf. on Autom. Plan. and Sched. (ICAPS 2018).* AAAI Press.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th Nat. Conf. on AI (AAAI 1996)*, 1194–1201.

Liang, J. H.; Oh, C.; Ganesh, V.; Czarnecki, K.; and Poupart, P. 2016. MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In *Proc. of SAT Competition 2016*. University of Helsinki.

Mali, A., and Kambhampati, S. 1998. Encoding HTN planning in propositional logic. In *Proc. of the 4th Int. Conf. on AI Plan. Systems (AIPS 2002)*, 190–198. AAAI.

Manthey, N.; Stephan, A.; and Werner, E. 2016. Riss 6 solver and derivatives. In *Proc. of SAT Competition 2016*. University of Helsinki.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *Journal of AI Research (JAIR)* 20:379–404.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Wu, D.; Yaman, F.; Muñoz-Avila, H.; and Murdock, J. 2005. Applications of SHOP and SHOP2. *Intelligent Systems, IEEE* 20:34–41.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 2014. Madagascar: Scalable planning with SAT. In *The 2014 International Planning Competition – Description of Planners*, 66–70.

Sinz, C. 2005. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proc. of the 11th Int. Conf. on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709, 827–831. Springer.

Soos, M. 2016. The CryptoMiniSat 5 set of solvers at SAT Competition 2016. In *Proc. of SAT Competition 2016*. University of Helsinki.

Straatman, R.; Verweij, T.; Champandard, A.; Morcus, R.; and Kleve, H. 2013. *Game AI Pro: Collected Wisdom of Game AI Professional.* CRC Press. chapter Hierarchical AI for Multiplayer Bots in Killzone 3.

Xie, F.; Müller, M.; and Holte, R. 2014. Jasper: The art of exploration in greedy best first search. In *The 8th Int. Planning Competition*, 39–42.

XPlan: Experiment Planning for Synthetic Biology

Ugur Kuter† and **Robert P. Goldman**† and **Daniel Bryce**† and **Jacob Beal**‡
Matthew DeHaven† and Christopher S. Geib† and Alexander F. Plotnick† and Tramy Nguyen‡ and Nicholas Roehner‡
†SIFT, LLC
Minneapolis, MN, USA
{dbryce, rpgoldman, ukuter, mdehaven, aplotnick, cgeib}@sift.net
‡Raytheon BBN Technologies
Cambridge, MA, USA
jacob.beal@ieee.org, nicholas.roehner@raytheon.com, tramy.nguy@gmail.com [*]

Abstract

We describe preliminary work on XPlan, a system for experiment planning in synthetic biology. In synthetic biology, as in other emerging fields, scientific exploration and engineering design must be interleaved, because of uncertainty about the underlying mechanisms. Through its experiment planning, XPlan provides a coordinating linchpin in DARPA's Synergistic Discovery and Design (SD2) platform to automate scientific discovery, closing the loop between multiple machine learning analysis and biological design tools and wet labs to guide the discovery and design process. To accomplish this, XPlan combines design of experiments techniques with hierarchical planning, based on the Shop2 planner, to develop experimental plans that are directly executable in highly automated wet labs and to project experimental costs. In particular, XPlan formulates experimental designs and translates them into goals representing biological samples, then uses Shop2 to plan construction and measurement of samples using available laboratory resources. In ongoing work, we are developing probability models that will support value of information computations to optimize experimental plans.

1 Introduction

In the field of synthetic biology, as with other emerging fields of engineering, scientific exploration and engineering design are intimately entwined. Unlike established fields of engineering, synthetic biology has only highly uncertain and incomplete mechanistic models. As a result, engineering synthetic biological systems is an incremental process in which the production of designs is closely interleaved with execution of experiments to assess the success of those designs and data analysis to identify factors and mechanisms responsible for design successes and failures.

Organization and planning of synthetic biology experiments is currently done almost entirely by hand. Several ongoing developments, however, are rapidly increasing the need for automation assistance in experiment planning. More and more laboratory automation is becoming available, increasing the scale and complexity of experiments that can be performed. Automation and information technology are supporting new business models with laboratory work done by technicians or outsourced to a "lab for hire." Finally, new "multiplexing" protocols allow many tests to be conducted on a single experimental sample, and multiple experimental samples to be processed in parallel. In all of these cases, the growth in scale and complexity are rapidly outstripping the abilities of humans to create detailed experimental plans and to hand-curate the relationships between those plans and the large collections of data they generate. Furthermore, experiments are still costly both in money and time, and the space a researcher wishes to explore is often much larger than the number of samples that can be tested, so there are opportunities for automation to assist in optimizing the value of information from each sample, potentially even dynamically based on partial results from an experiment in progress.

This paper explains how our XPlan planner, based on Hierarchical Task Network (HTN) planning, addresses these issues by providing automation support for experiment planning. In the next section, we describe the class of discovery and design problems addressed by XPlan, and the challenges they pose. We then explain how our HTN approach, based on the Shop2 planner, addresses these challenges, and describe our early-stage work on optimizing the expected value of information while planning experiments. Finally, we summarize and describe some next steps.

2 Synergistic Discovery and Design (SD2)

Synthetic biology is the systematic engineering of living organisms to perform desired functions. For example, biological sensors have applications in sensing biological, chemical, and radioactive weapons, and pathogens; effectors have applications in chemical synthesis and cleanup, and targeted medical therapies. Because existing models for genetic structures, assembly, and expression are still relatively weak, however, synthetic biology necessarily involves both design and experimentation to assess the success of designs and identify factors responsible for success and failure.

DARPA's Synergistic Discovery and Design (SD2) program seeks to speed scientific and design processes through automated support for experiment planning, automated execution of experimental protocols across laboratories, and

[*]This work was supported by the Air Force Research Laboratory (AFRL) and DARPA under contract FA875017CO184. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL and DARPA.

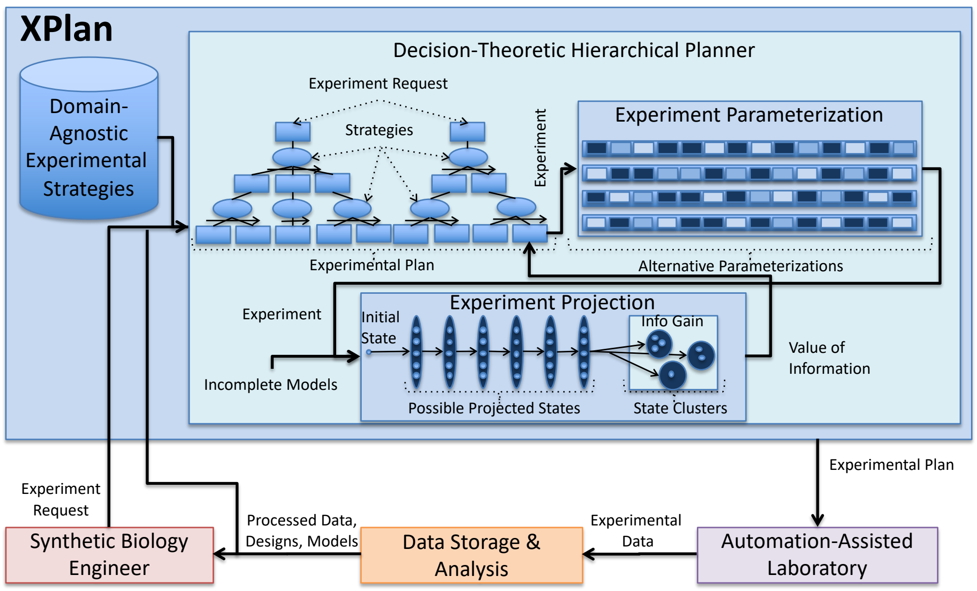
XPlan
Domain-Agnostic Experimental Strategies
Decision-Theoretic Hierarchical Planner
Experiment Request
Strategies
Experiment
Experiment Parameterization
Experimental Plan
Alternative Parameterizations
Experiment
Experiment Projection
Initial State
Info Gain
Value of Information
Incomplete Models
Possible Projected States
State Clusters
Experimental Plan
Experiment Request
Processed Data, Designs, Models
Experimental Data
Synthetic Biology Engineer
Data Storage & Analysis
Automation-Assisted Laboratory

Figure 1: XPlan combines domain-agnostic strategies and domain-specific knowledge to expand experiment requests into executable plans, which are then parameterized and projected for VOI analysis. Plans are given to laboratories to execute, producing data that results in updated models and designs and new experiment requests.

high-speed, large-scale exploratory data analysis. Figure 1 shows the high-level architecture of XPlan, our hierarchical experiment planning system, which is a key part of the overall SD2 project. XPlan uses HTN planning to generate experimental protocols from synthetic biologists' expressions of experimental intent. It also translates the protocols it generates into executable forms so that they can be performed at different laboratories, which have different equipment, levels of automation, and processes.

XPlan also helps with data analysis, by storing information about the protocol in SynBioHub (McLaughlin et al. 2018; Madsen et al. 2016; Roehner et al. 2016), a standard synthetic biology semantic database. This enables labs performing protocols to accurately and consistently label the resulting data, immensely simplifying the process of data analysis. It also enables the operation of an automated pipeline for preliminary data formatting, labeling, and processing.

Experiment Planning for synthetic biology is challenging for a number of reasons:

- **Complex Systems and Incomplete Models:** The causal processes underlying biological mechanisms and their response to external stimuli are complex and only partially known. Designing biological circuits is difficult because of constraints such as the interactions of elements of the design with one another, and with existing biological functions (e.g., the design takes resources the cell needs to live). Therefore, decisions such as how long to incubate samples (to allow cells to multiply and respond to their environment) are difficult to make. Such decisions are made based upon the expertise of biologists without a formal causal model that would support the simulation/projection necessary for first-principles planning.
- **Replication:** Replicating experiments at multiple locations is critical in emerging fields. This follows from the incomplete knowledge described above: since we are not certain what environmental factors are most critical, and must be most carefully controlled, replication can provide even more information than in well-understood fields.
- **Large Sample Sizes:** Recent advances in biotechnology have reduced experiment costs through increased parallelization. In many cases, the cost of culturing hundreds of samples is only marginally more than the cost of a single sample. However, some operations such as sequencing the genome of a sample are not parallelizable. This is a challenge for planners because they must reason about many samples (objects). In some cases operations apply to all samples and do not impact the plan search branching factor. In other cases, the planner must select between subsets of samples for an operation, and hence cope with a large branching factor.

The problem of planning experiments for synthetic biology has several interesting features from a planning perspective:

- **Long Planning Windows:** Experiments often execute over the course of multiple days, providing a long window between planning episodes. Each planning episode involves selecting a new set of samples and conditions to investigate based upon the outcome of previous experiments. With several days to plan, it is possible to consider many possible plans. This allows for a more costly analysis of alternatives and shifts the emphasis from finding satisficing plans to finding high quality plans.
- **Multiple Levels of Abstraction:** Plan executors (laboratories) offer a range of robot and human executed primi-

tive actions. Each laboratory offers a layer of abstraction over the actions that go into experimental protocols, and those layers of abstraction vary between labs based on the extent of automation, equipment available, and management structure. This provides an interesting relationship between choice of performer and nature of the procedure. Also, by providing a common view on these differing institutions, XPlan can give real value to its users.

- **Managing Costs and Benefits:** Synthetic biology experiments are expensive, and because different labs' cost models are different, it can be difficult for biologists to predict the costs of performing a particular protocol at a particular facility or set of facilities. XPlan incorporates multiple labs' cost models into its HTNs to compute costs along with protocols. To help with the benefit side of the analysis, we are beginning to add *value of information* computation and guidance to XPlan to help biologists get the most useful information with their limited resources (both monetary and human).

3 Experiment Planning

Experiment planning involves two sub-problems: (1) experimental design, to select the samples and conditions to test, and (2) plan synthesis, to create the procedures that will construct and then measure the samples. In this paper, we focus on plan synthesis and describe our hierarchical planning approach based on our Shop2 system (Goldman and Kuter 2018; Nau et al. 2003; 2005).

Shop2's hierarchical planning approach is particularly well-suited to planning synthetic biology experiments. HTNs enable us to easily capture expert knowledge from biology researchers and formalize that knowledge for use in planning. A specific advantage of Shop2 is that it is a forward state progression planner: it performs task decompositions in the order those tasks will be executed in the world, while progressing the current state. Because it does forward state space planning, Shop2 always has a full model of the current state of the world (and the history that led to it). This full world state enables Shop2 to incorporate considerably greater expressive power – for example, capabilities for calling attached procedures, making axiomatic inferences, and performing numeric computations – than other HTN planners (e.g., UMCP (Erol, Hendler, and Nau 1994) and Sipe (Wilkins 1988)), that work with partial world models. Shop2 was developed this way for work on designing for manufacturability, which involved using CAD tools in projective planning. Such tools require full state descriptions, and typically are incapable of regression. For example, one cannot look at a machined part and reason to what was used to build it, but it's straightforward to give a blank and a design to a CAD tool to identify the required cuts and project the resulting part. Progression search also *potentially* allows for easier incorporation of informed heuristics, though at the moment the heuristics in XPlan are incorporated in the HTN preconditions. It is an open research question how to combine informed heuristics with such expressive preconditions, and with the task-based, as opposed to goal-based, semantics of Shop2 plans.[1]

XPlan's plan library is divided into three components, broadly speaking. First is a high-level library of experimental strategies that is not specific to synthetic biology or to particular laboratories. These strategies aim to distribute experiments across laboratories for execution while minimizing variation, validating hypotheses, and determining parameters for designs during planning. Second is an abstract set of protocol components that are specific to synthetic biology, but not to particular lab configurations. Finally, there are methods that are specific to particular labs, and that enable our procedures to be translated into executable form. For example, some of these library components enable an XPlan-generated experimental protocol to be translated to Autoprotocol. Autoprotocol, developed by Transcriptic[2], is an executable JSON schema providing a domain specific language for automated wet lab operation.

Consider an experiment for measuring growth rate of a set of modified yeast strains over time via optical density (OD), which characterizes the amount of cells interfering with light shining through a sample. Experiment plans must first select the combination of biological factors that will be most informative. Examples for yeast strains include the modified genes, the yeast strain itself, small molecule concentrations, and other environmental factors (media type, temperature, humidity, etc.). These factors affect growth rate, which can be estimated across time by monitoring changing OD.

Listing 1 shows a Shop2 planning operator for the process of "provisioning" a replicate, i.e., collecting a sample from a particular strain of micro-organism (identified by `?resource`) in order to use it as a replicate in an experiment. A "replicate" is one of multiple copies of the same strain/conditions pair, used to ensure results are not lost due to mistake, and to provide sufficient data for later analysis. Intuitively, "provisioning" like selecting a cup of an ingredient for use in a recipe. The resource argument will be bound to a URI pointing to a SynBioHub entry representing a strain of yeast.

Listing 1: Example Shop2 sample provisioning operator definition.

```
(:op (!provision-replicate ?sid
                ?resource ?colony-type)
 :precond
   ((experiment-id ?ex-id)
    (assign ?sample-uri
      (make-sample-uri ?ex-id ?sid)))
 :add
   ((experiment-sample ?sid ?sample-uri)
    (derived-from ?sid ?resource)
    (resource ?colony-type ?sid)
    (sample-map
       (:source ?resource
          :destination ?sample-uri)))
  :cost 0.0)
```

This operator description highlights one of the expressive features of Shop2's HTNs not present in traditional planners: its ability to make external function calls during planning and incorporate the return values of those calls into its plan and

[1]See Goldman (2009), for a discussion of the semantics of Shop2 task networks.

[2]`https://www.transcriptic.com/`

state. In the above example, the operator will call the function `make-sample-uri`, passing the values of two variables (the experiment id and the sample id) from the plan space as arguments and receive a newly-generated URI for the sample. Unlike classical or other HTN planners, the ability to make external function calls makes Shop2 Turing-complete[3] and highly applicable to practical planning domains.

We also use Shop2's facilities for computing action costs to compute experiment costs. Shop2 allows an author to specify either static or dynamic cost functions in operator descriptions. The former is a fixed number across all instantiations of the operator description (e.g., the operator in Listing 1 has a static zero cost). The latter defines cost value as a function of the parameters from preconditions and task arguments. Listing 2 shows an operator whose cost is computed by looking up lab-specific costs for a growth method (`?meth`) and the lab's minimum sample size. This cost summary is then accumulated to compute the cost of a plan—in this case, the cost of an experiment. We define the cost of an experiment based on propriety information gathered from specific laboratories. This information includes both monetary and human costs.

Listing 2: Example Shop2 cost computation.

```
(:op (!!calc-culture-cost ?performer ?meth)
    :precond
      ((cost ?meth
             ?performer ?cost-per-sample)
       (min-sample ?meth ?performer
                   ?number-of-samples))
    :cost (* ?cost-per-sample
             ?number-of-samples))
```

Listing 3 shows an example Shop2 method for provisioning yeast colonies into samples to be used in an experiment. This is a recursive HTN method in Shop2's language; it enables the planner to iterate over the yeast colony ?resources given in the first argument to the head task and provision a replicate sample for each of those resources. Unlike traditional planning model languages, Shop2 allows sets as possible values for variables in a method or operator. For example in Listing 3, the variables ?resources and ?provisioned-samples hold lists of yeast colony descriptions.

Listing 3: Example Shop2 method for sample provisioning.

```
(:method
 ;; Head task:
 (provision-all-resources ?resources
                    ?provisioned-samples)

 ;; preconditions for recursion base case:
 ((= ?resources nil)
  (bagof ?map (provision-sample-map ?map)
        ?sample-map))
 ;; subtasks(s) for the base case:
 (!provision :name ?name
   :transformations ?sample-map)

 ;; Recursive step
 ;; preconditions
 ((= ?resources (?resource . ?rest)))
 ;; subtasks
 ((provision-replicates ?resource
         ?type ?replicates)
  (provision ?rest ?provisioned-samples)))
```

[3]See Appendix A.

The first precondition specifies the base case: checking to see if all of the resources have been provisioned, i.e., if the `?resources` list is empty. If so, the planner collects the entries of the provision sample map. Shop2's preconditions language includes Prolog's `bagof`, which finds all bindings to a variable in a logical formula and collects them. Listing 3 collects the values of the `?map` variable from every grounded `provision-sample-map ?map` logical expression in the current state. Next Shop2 will invoke the `!provision` operator with information from the sample map. In the recursive branch, the preconditions specify that the resources list must not be empty and split it into a first resource and the rest of the resources. The subtasks when this match are to provision the replicates for the first resource, and then recursively handle the remaining resources.

4 Value of Information

Providing predictable cost information was one of our sponsor's highest initial priorities: they are very concerned with facilitating scientific discovery by making labs-for-hire easier, more transparent, and more cost-effective to use. As we described above, XPlan can already provide estimated costs for performing protocols at multiple labs. To go beyond this and provide further support to users, we are adding techniques, based on *value of information* (VOI), to estimate the benefit of particular protocols, so that XPlan (likely in collaboration with its user) can guide users to more informative experimental protocols. We also hope that the analysis we conduct in the process will shed light on questions such as "how many biological and technical replicates are appropriate?", "how important is it to test *this* design across multiple laboratories?", and "how many tests are necessary to build confidence that a design is reliably replicable?"

In conventional decision analysis, VOI is defined as the difference between the expected utility of a decision made with a particular piece of information, and without that information (Pearl 1988, Chapter 6)). In design problems proper, we can use the estimated value of a successful design to compute the value of information that contributes to the design. For cases where the design is not directly useful (today many designs are made for exploratory reasons, not for employment), we will take the information produced (in terms of information distance between prior and posterior) as a proxy for utility. Unfortunately, VOI is notoriously difficult to compute (Krause and Guestrin 2009), because it requires reasoning about multiple possible outcomes of experiments.

Our work on this part of XPlan is at a very early stage, but we can characterize the direction we are taking. We expect to use Monte Carlo Tree Search (MCTS) to approximate the value of information (Kamar and Horvitz (2013) use this technique but in a much simpler problem). Since the information-gathering process will be driven by execution of experimental processes, we will use Shop2 to build the trees for the MCTS. We are still working out the extent to which the problem will involve conditional planning – typically there is little closed-loop control of the experimental protocol based on the information produced. That information is generally extracted in an offline data analysis process after the protocol is completed. Closed-loop control is typically limited to correcting failures. If true, that will simplify the construction of the protocol significantly, and avoid the need to generate a large and complex experimental *policy* instead of an experimental *plan*. That said, XPlan will still have to explore many branches to find the VOI of alternative plans.

5 Conclusions and Future Work

Our XPlan system uses HTN planning in an interesting new domain: experiment planning for synthetic biology. XPlan exploits the expressive power of the Shop2 planner to handle many of the challenges in coupled engineering design and scientific exploration of emerging fields. It adapts to domains with weak mechanical models, in a way that would be difficult, if not impossible for first principles planners. It has already shown utility by computing experiment costs across different labs, and by automating the process of aligning experimental data with experimenter intent in ways that enable the automation (and hence the speed-up) of data analysis. The fact that XPlan's plans can be compiled into executable procedures will provide value in the near future, as the SD2 pipeline is completed. In ongoing work, we are extending our cost modeling to incorporate benefits – in terms of VOI.

A Shop2 is Turing-complete

Since Shop2 can invoke arbitrary functions in its preconditions (Nau et al. 2003), it can invoke a Turing machine simulator as an external function, and have a task network that would take a universal Turning machine program as parameter and return a plan iff that program terminates. □

References

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *Proc. Second International Conf. on AI Planning Systems (AIPS-94)*, 249–254.

Goldman, R. P., and Kuter, U. 2018. Explicit stack search in shop2. Technical Report SIFT-TR-2018-1, SIFT, LLC, Minneapolis, MN.

Goldman, R. P. 2009. A semantics for HTN methods. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *ICAPS*. AAAI.

Kamar, E., and Horvitz, E. 2013. Light at the end of the tunnel: A Monte Carlo approach to computing value of information. In Gini, M. L.; Shehory, O.; Ito, T.; and Jonker, C. M., eds., *AAMAS*, 571–578. IFAAMAS.

Krause, A., and Guestrin, C. 2009. Optimal Value of Information in Graphical Models. *Journal of Artificial Intelligence Research* 35:557–591.

Madsen, C.; McLaughlin, J. A.; Mısırlı, G.; Pocock, M.; Flanagan, K.; Hallinan, J.; and Wipat, A. 2016. The SBOL Stack: A Platform for Storing, Publishing, and Sharing Synthetic Biology Designs. *ACS Synthetic Biology* 5(6):487–497.

McLaughlin, J. A.; Myers, C. J.; Zundel, Z.; Mısırlı, G.; Zhang, M.; Ofiteru, I. D.; Goñi-Moreno, A.; and Wipat, A. 2018. SynBioHub: A Standards-Enabled Design Repository for Synthetic Biology. *ACS Synthetic Biology* 7(2):682–688.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, J. W.; Wu, D.; and Yaman, F. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20(2):34—41.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Waltham, MA: Morgan Kaufmann.

Roehner, N.; Beal, J.; Clancy, K.; Bartley, B.; Misirli, G.; Grünberg, R.; Oberortner, E.; Pocock, M.; Bissell, M.; Madsen, C.; Nguyen, T.; Zhang, M.; Zhang, Z.; Zundel, Z.; Densmore, D.; Gennari, J. H.; Wipat, A.; Sauro, H. M.; and Myers, C. J. 2016. Sharing Structure and Function in Biological Design with SBOL 2.0. *ACS Synthetic Biology* 5(6):498–506.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers.