T-HTN: Timeline Based HTN Planning for Multiple Robots

Viraj Parimi[1], **Zachary B. Rubinstein**[2], **Stephen F. Smith**[2]

[1] Massachusetts Institute of Technology, [2] Carnegie Mellon University

Abstract

Effective coordinated actions by a team of robots operating in close proximity to one another is an important requirement in many emerging applications, ranging from warehousing and material movement to the conduct of autonomous housekeeping and maintenance of deep space habitats during unmanned periods. Yet, such multi-robot planning problems remain a significant challenge for contemporary planning technologies, due to several complicating factors: goals must be assigned to robots and accomplished over time in the presence of complex temporal and spatial constraints in a manner that optimizes overall team performance, attention must be given to the durational uncertainty inherent in robot task execution, and planning must be responsive to changing and unexpected execution circumstances. In this paper, we present `T-HTN`, a novel planner that attempts to overcome this challenge by coupling the structure and efficiency of Hierarchical Task Network (HTN) models with the flexible scheduling infrastructure of timeline-based planning systems. We present initial results on a simple set of multi-robot problems that show the potential of `T-HTN` in comparison to a state-of-the-art PDDL-style temporal planner.

Introduction

Generation of multi-robot plans that optimize overall team performance in the presence of tight temporal-spatio constraints remains a significant challenge for contemporary automated planning frameworks. On one hand, heuristic, action-based temporal planners (both PDDL-style (Coles et al. 2010; Do and Kamhbampati 2003; Eyerich, Matmuller, and Roger 2009) and HTN-style (Qi et al. 2017)) typically achieve tractability by constraining the plan generation process to forward state-space search (i.e., expanding plans in a strict time-forward order). This assumption can be quite awkward for satisfying certain types of temporal constraints (e.g., task deadlines), and more generally can be quite limiting with respect to optimizing team performance objectives across a set of goals (e.g., minimizing makespan, maximizing the number of goals satisfied). On the other hand, timeline-based planners (e.g., (Muscettola et al. 1992, 1998; Fratini, Pecora, and Cesta 2008; Verfaillie and Lematre 2003; Umbrico et al. 2017) provide a flexible, Simple Temporal Network (STN)-based infrastructure (Dechter, Meiri, and Pearl 1991) that allows planning actions to be inserted opportunistically at various time points across an agent's planning horizon during plan generation so as to better optimize global properties of the plan (such as its makespan). But timeline-based systems lack general principles for global search control, and tend to be driven by use of hand-crafted, domain specific heuristics that do not transfer easily to new problems.

Drawing inspiration from previous work in constraint-based scheduling (Smith, Becker, and Kramer 2004; Rubinstein, Smith, and Barbulescu 2012), wherein global search control revolves around allocation of resources to instantiated task networks, this paper proposes a multi-agent planning and scheduling framework that attempts to combine the strengths of both action-based and timeline-based planning approaches, and describes its initial implementation in a prototype planning system called `T-HTN`. To overcome the inefficiency and idiosyncrasy of reasoning about actions at the individual component level, `T-HTN` couples timeline-based reasoning with an HTN planning model designed for resource allocation. By introducing hierarchical structure that organizes task decomposition around allocation of resources to tasks and placement of their constituent actions on resource timelines, `T-HTN` is able to achieve both search efficiency and flexibility, while retaining the representational expressiveness and generality of state-of-the-art temporal planning frameworks.

Some previous efforts have attempted to exploit the structure imposed by an HTN planning model within a rich temporal reasoning infrastructure. In (Qi et al. 2017), the SHOP2 planner was extended to include a temporal reasoning component and provide the ability to generate plans with durative actions. However SHOP2's reliance on forward plan-space search was retained to avoid the added complexity of timeline-based reasoning, and hence the issue of plan quality in the case of multiple goals and actors was not addressed. On the timeline-based planning front, recent work by (Umbrico et al. 2017) has also proposed the use of hierarchical structure as a means of directing search control. However, this framework is rooted in the dynamical systems origin of the timeline-based planning paradigm of modeling and controlling a system of physical components over time, and hence their notion of hierarchy focuses only on aggregate physical system structure. Perhaps closest in spirit to our approach is the recently introduced FAPE planner (Bit-Monnot et al. 2020), which also aims at integrating

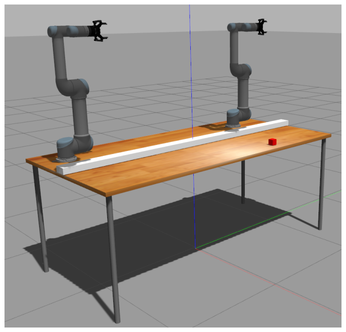

Figure 1: Consider two UR5 robotic arm manipulators mounted on shared railway network who are tasked to move the red cube from its initial location to a target location within a specified deadline.

HTN and timeline-based planning to gain search efficiency. However, FAPE is designed to accommodate a mix of generative and HTN planning. It incorporates a mix of search control strategies to this end, and in the case of a fully specified HTN model (our interest in this paper), FAPE falls back on the same forward heuristic search algorithm that limits other HTN planners. `T-HTN`, in contrast, retains the flexibility to opportunistically expand the developing plan and centers plan expansion around allocation of resources to achieve search efficiency.

In this paper, we summarize the `T-HTN` planner, and present initial experimental results on a set of simple multi-robot planning problems that show its potential in comparison to POPF (Coles et al. 2010), a state-of-the-art action-based temporal planner.

Running Example

Figure 1 introduces a simple problem, motivated by our research interest in robotic systems for autonomous maintenance of deep space habitats, that we will use as a running example throughout the paper. In this scenario, two UR5 robotic manipulators are mounted on a shared railway network. The network is divided into *rail blocks* and two ensure safe operation, only one robot can occupy a given rail block at a time. Three attributes are used to model the state of each robot at any point in time: the rail block it is occupying, the position of its arm and the state of its gripper. Initially, each robot arm starts in the "home" (contracted) position, which is required for rail travel, and each arm's gripper state is empty. Each arm has primitive actions for travelling across rail blocks (*rail_move*), for grasping (*grasp*) and releasing (*release*) objects, and for returning the arm to its home position (*move_to_home_state*). Figure 2 shows a *move-item* HTN for the example problem.

Technical Approach

Like all timeline-based planning systems, `T-HTN` replaces the classical planning representation of current state as a forward-rolling collection of facts with an explicit characterization of its evolution over time, where various aspects

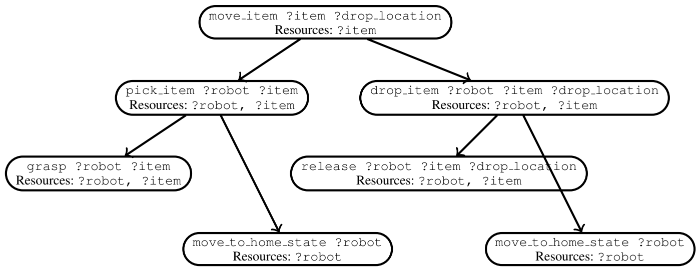

Figure 2: Compiled HTN decomposition tree for the working example as described in Figure 1

of state (referred to as *state variables*) are represented as sequences of facts (or *state values*) that are true over time. By definition, this change results in a greatly expanded search space vis a vis traditional HTN planning formalisms, as states corresponding to specific intervals across the planning horizon must now be examined to situate actions in the plan. To gain search efficiency in this expanded search space, `T-HTN` introduces and exploits additional problem structure. Specifically, we ascribe special status to objects declared as resources and assume that all primitive (leaf) nodes in a HTN plan (referred to as *actions*) will ultimately be allocated to and scheduled on the specific resource(s) that will carry them out. Accordingly each declared resource has an associated state variable that represents the resource's availability over time, referred to as its *timeline*. When an action in the plan is scheduled on a resource r, an *action token* is inserted onto r's timeline for the determined interval. This action token indicates all aspects of the action's state that are true at the start, end or throughout the scheduled interval, and the key assumption is that these aspects of state will change only as new actions are subsequently inserted before or after this action token on r's timeline. The insertion of an action token on r's timeline may also result in derivative state changes to other timelines (e.g., the effects of a *grasp* action will dictate the start of a change to the location of the grasped item). To allow for specifying this additional resource structure, `T-HTN` uses an extended version of HDDL (Holler et al. 2020) for representing the domain and problem inputs.

Language Extensions

Using a model similar to the one used in the Action Notation Modeling Language (ANML) (Smith, Frank, and Cushing 2007) for representing resources and complex objects, `T-HTN` uses types to designate objects as resources. In this paper we restrict attention to resources of type discrete_reusable_resource. The aspects of state that are stored with an action token on a given resource's timeline are specified as an optional second form on the type definition that denotes variable-type pairs (delimited by matching curly brackets). When objects of a resource type are created, relevant aspects of state are initialized by providing an optional second argument (also delimited by curly brackets and in the form of *predicate = value*). As new action tokens are inserted onto a resource timeline, the preconditions and effects of the corresponding actions will dictate how relevant aspects of state will change. The following snippet is an example definition of `robot` and `rail_block` resource types and initialization of a pair of robots:

```
Definition:
    (:types
        robot {?block - rail_block
            ?arm-position - state
            ?gripper-state - item} -
                discrete_reusable_resource
                    )
 Initialization:
    (:objects
        ur5A, ur5B - robot)
    (:object-instances
        ur5A {block = blockA,
            arm-position = home, gripper
                -state = empty}
        ur5B {block = blockD,
            arm-position = home, gripper
                -state = empty})
```

A second extension that `T-HTN` makes to HDDL involves specification of temporal constraints. In this case, we borrowed directly from PDDL 2.1, incorporating its `:duration` field and the temporal qualifiers (`atstart`, `atend`, `overall`) used in preconditions and effects into the action definition of HDDL. For example:

```
(:action move_to_home_state
    :parameters (?r - robot)
    :duration (= ?duration 10)
    :precondition (and
        (atstart (unsafe state
            ?r.arm-position)))
    :effect (and
        (atend (= ?r.arm-position
            home)))
)
```

By definition, any resource that is allocated to an action is unavailable `overall` of the action's scheduled interval.

A third extension to HDDL allows for the use of specialized algorithms for handling specific planning sub-problems efficiently. In our running example, algorithms for solving the multi-agent path-finding problem (MAPF) are relevant to the movement of robots along the rail network in service of tasks in a collision-free manner. Rather than modeling the movement possibilities as adjacent blocks on the rail and trying all combinations until a goal location is found, application of a specialized planner can quickly determine an efficient route by coupling a shortest-path algorithm with a collision-avoidance strategy. This extension is achieved in `T-HTN`with two additional constructs: (1) the definition of a functional predicate with `f-predicate` and (2) the definition of functional methods with `f-method` keywords. Using these constructs, one can tie specialized external planners to the domain representation based on their unique name identifiers. Functional methods are slightly different in their representation than standard methods in terms of their defined task network. Since they are connected to a specialized external planner, we provide a simple wrapper function whose name is restricted to be the same as the corresponding functional method. Its purpose is to first convert the input from the parsed format to the API that the specialized planner supports and then to convert the output of the specialized planner into a sequence of actions and temporal constraints that it can be linked into the overall task network. The following snippet shows a specification of these constructs for the MAPF planner in our example scenario:

```
Predicate:
    (:f-predicates
        (clear ?from ?to - rail_block)
    )

Method:
    (:f-method m_clear_and_move
        :parameters (?r - robot
            ?to - rail_block)
        :task (clear_and_move ?r ?to)
        :precondition (and
            (atstart (not (clear
                ?r.block ?to))))
        :effect (and
            (atend (clear ?r.block ?to))
                )
    )
```

A final extension to HDDL allows for specifying a broader set of temporal constraints between tasks. HDDL allows specification that tasks must be done in a particular order for example, but does not allow specification that a task must be done immediately after another task completes. To address this limitation, `T-HTN` introduces the `:sync-constraints` field to the `:method` construct, in which any set of pairwise constraints can be added. Using built-in constraint types, this construct supports general specification of Allen temporal relations between tasks. In the following example using `:sync-constraints`, *meets* denotes that `task1` must start immediately after `task0` completes.

```
(:method m_pick_item
    ⋮
    :ordered-subtasks (and
        (task0 (grasp ?r ?i))
        (task1 (move_to_home_state ?r)))
    :sync-constraints (and
        (task0 meets task1))
)
```

Core Search Procedure

Leveraging the domain representation, `T-HTN` employs an incremental algorithm for generating and feasibly inserting a new task plan into the current global multiagent plan/schedule. It first enumerates all possible decompositions of the incoming request and then instantiates a task network for each decomposition that is tied to an underlying STN. Each possible task network instantiation still needs to be grounded with a specific set of resource assignments, and the choice of resources can significantly affect overall plan quality. Alternative sets of resource assignments for a given instantiated task network are explored by applying a backward timeline scanning procedure to each. For a given set of resource choices, the scanning procedure is applied to determine the set of slots on relevant resource timelines where actions can be feasibly scheduled. We believe this ability to (1) organize the search around alternative sets of resource choices and (2) exploit timeline structure to determine feasible options is key to achieving overall planning/scheduling efficiency in multi-agent domains.

Enumerating Decompositions As mentioned before, `T-HTN` uses the standard HTN task decomposition process by methods to produce an AND/OR HTN that represents alternative solution paths. Once the path decomposition tree is generated, the next step is to enumerate all the possible decompositions, by combinatorically expanding the existing OR nodes. Algorithm 1 provides an efficient, high-level, recursive algorithm that enumerates all paths in the tree while guaranteeing that no potential alternatives are skipped. This algorithm follows from a straightforward depth-first search procedure where, at each level, we keep track of the choices made thus far and then recursively maintain a cartesian product of all such decisions. The worst-case time complexity of Algorithm 1 is dominated by the size of the cartesian product, which is computed in Line 15.

Algorithm 1: Algorithm to return all the possible instantiations of a given path decomposition tree.

```
 1: procedure ENUMERATE DECOMPOSITIONS(vertex)
 2:     Initialize empty leafs vector
 3:     if vertex is a leaf then
 4:         Add vertex to leafs
 5:         return leafs
 6:     else if vertex is OR then
 7:         for all children c of vertex do
 8:             leafs +=
                    ENUMERATE DECOMPOSITIONS(c)
 9:         end for
10:     else
11:         Initialize empty op vector
12:         for all children c of vertex do
13:             op += ENUMERATE DECOMPOSITIONS(c)
14:         end for
15:         leafs = cartesian_product(op)
16:     end if
17:     return leafs
18: end procedure
```

Instantiating Alternative Task Networks For each possible alternative decomposition in turn, `T-HTN`instantiates a task network that is tied with the underlying STN and contains tokens corresponding to constituent tasks and actions. Each token designates a start and end time point for a particular task/action, and, if the token corresponds to an action, the corresponding duration constraint is enforced between the start and end time points. Plans are generated for a task network by scheduling its action tokens on the timelines of compatible resources. An action token is scheduled on a compatible resource timeline by searching for *slots*, i.e., temporal intervals of availability between tokens on the timeline, in which the action token can be feasibly inserted.

Algorithm 2: Algorithm to return an instantiated task network (connected to the underlying STN) that enforces all pre-specified temporal constraints.

```
 1: procedure EXPAND(vertex, tree, STN)
 2:     Create a token for vertex
 3:     Connect token to tree
 4:     if vertex is not leaf then
 5:         for all children c of vertex do
 6:             EXPAND(c, tree)
 7:         end for
 8:         Add synchronization constraints to the STN
 9:         Add contains constraints to the STN
10:     end if
11:     return tree
12: end procedure
13: procedure INSTANTIATE TASK NET(search, STN)
14:     Instantiate an empty tree
15:     tree ← EXPAND(search.root, tree, STN)
16:     Add the release time constraint to search.root
17:     Add the due date constraint to search.root
18:     return tree
19: end procedure
```

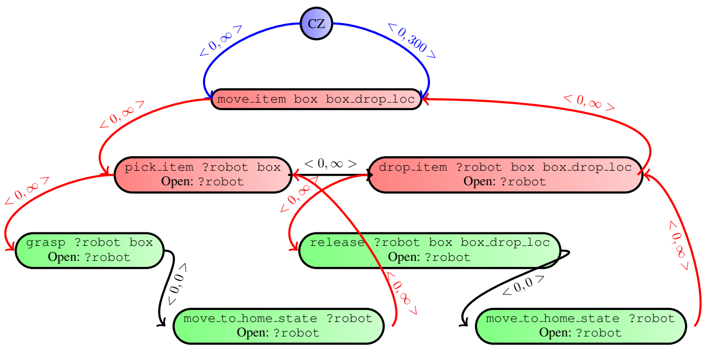

Figure 3: Instantiated task network for the working example as described in Figure 1. Nodes in red represent the high-level tasks, and the nodes in green represent the action primitives. The blue edges correspond to the release time and due date constraints, while the red edges correspond to the contains constraint. The black edges correspond to the pre-specified synchronization rules.

`T-HTN` enforces a *contains* temporal constraint between each higher-level token and its corresponding decompositions so that the two levels are temporally linked. Such constraints ensure that any temporal constraints imposed on an aggregate task are also applied to its constituent sub-tasks. If the constituent sub-tasks of an aggregate task are known to be *ordered*, then the *contains* constraint need only be enforced between the start time point of the parent task to the start time point of the first task in ordered decomposition and the end time point of the last sub-task in the ordered decomposition to the end time point of the parent task. Absolute temporal constraints, such as release times and due dates, are asserted to corresponding root task and propagated down the network via the contains constraints. To efficiently compute all resource and parameter assignments, we segregate choices based on whether they have been already made or not. Decisions that have been made prior are moved to a *closed* set, while the decisions that remain to be made are moved to the *open* set. A cartesian product is then computed on all the possible assignments of *open* set parameters and these are used to ground the generated task network for scheduling. Algorithm 2 provides a high-level overview of how task network generation proceeds. Assuming that there can be V nodes in the generated tree, the worst-case time complexity of the algorithm is $O(V)$ since each vertex needs to be visited at least once so that it can be part of the task network. Figure 3 shows the generated task network for the working example as described in Figure 1.

Finding slots Once the instantiated task network is built, it acts as a template for all possible resource and parameter assignments. For each possible combination of assignments, `T-HTN` determines the set of feasible slots by a backward timeline scanning process that repeatedly attempts to backward schedule the action tokens in the instantiated task network at each possible start point. The timeline scanning process iterates through all required resource timelines in reverse order, identifying sequential pairs of tokens currently on the timeline that delineate potential slots. The search of slots pivots around resources of type *robot*, while considering other required resources (e.g., rail_blocks) as dependent resources. The action tokens of the instantiated task network are only scheduled on *robot* and *rail-block* timelines.

At the same time, the effects of an inserted action may also imply changes to other, dependent state variable timelines. For each combination of slots identified on a resource timeline and its dependent state variable timelines, `T-HTN` queries the underlying STN to confirm the feasibility of that slot based on current temporal bounds. This process helps prune infeasible combinations of slots early on, and thereby speeds up the scanning process. Following these checks, `T-HTN` constructs a coherent world state of the environment by iteratively modifying the initial world state with the effect literals encapsulated by the currently considered slots. This updated world state is utilized to check the preconditions of actions, and assuming that all preconditions are verified, the temporal constraints imposed on the action token corresponding to the action are enforced in the underlying STN. This includes retraction of the sequencing constraint between the two tokens on the timeline surrounding the current slot, and the posting of two new sequencing constraints to insert the new action token into this slot. If all of these constraints can be consistently asserted, the slot is a feasible assignment for the action token. `T-HTN` uses this procedure to generate sets of feasible slot assignments (options) for all action tokens in an instantiated task network by trying to schedule each of the possible combinations of assignments in their Cartesian product. The maximum number of options generated before terminating the search can be restricted by setting a customizable variable to the preferred number. Finally, once all options have been generated, they are evaluated according to some set of objective criteria (e.g., minimize overall task makespan, complete task as early as possible, reduce disruption to plans of other robots), and `T-HTN` commits to the best option.

Algorithm 3: Algorithm to find a set of feasible slots for a given instantiated task network while attempting to satisfy any failing precondition literals.

```
 1: procedure SATISFY PRECONDITION(lit, task_net, STN)
 2:     Find the satisfying task tk
 3:     Identify tk's required resources and parameter assignments
 4:     if tk is an action primitive then
 5:         Create token corresponding to tk
 6:         Add token to task_net
 7:         Add temporal constraints of token to STN
 8:     else
 9:         tokens ← Call the specialized external planner
10:         for all token in tokens do
11:             Add token to task_net
12:             Add temporal constraints of token to STN
13:         end for
14:     end if
15: end procedure
16: procedure FIND SLOTS(task_net, STN)
17:     Instantiate an empty slots data structure
18:     for all Leaf tasks tk in task_net.leafs do
19:         Collect tk's required resource $\mathcal{R}$ assignments
20:         for all Slots s over $\mathcal{R}$ do
21:             Check temporal bounds of s using STN
22:             Compute the world state ws using s
23:             for all Precondition literal p over tk do
24:                 if p fails against ws then
25:                     SATISFY PRECONDITION(p, task_net, STN)
26:                 end if
27:             end for
28:             Enforce temporal constraints of tk onto the STN
29:             slots += s
30:         end for
31:     end for
32:     return slots
33: end procedure
```

Note that sometimes it is hard to encapsulate the entire precondition check within single or multiple literals. By utilizing the functional predicate constructs introduced earlier, we use specialized algorithms to compute such prerequisite checks efficiently and optimally where appropriate[1]. If, however, any precondition literal fails either via a functional

[1]In our current context, this refers to the MAPF solver mentioned earlier.

predicate call or via a violation of a constant literal in the process of checking preconditions against an updated world state, we attempt to satisfy such failing preconditions in two different ways. First, we iterate through the list of actions and identify potential actions whose effects match the failing precondition. If we find such an alternative, the newly instantiated tokens are then added to the same task network, and the procedure continues normally. Second, we iterate through the list of functional methods and identify potential solutions whose effects match the failing precondition. If we find such a method, we call the specialized planner associated with that functional method. We update the task network by adding all newly generated action tokens, after which the procedure continues normally. Whenever `T-HTN` finds an alternative action or method, it validates that alternative by comparing its set of preconditions with the updated world state. If there is any violation, `T-HTN` continues to look for other alternatives until they are exhausted. Since this process can potentially lead to infinite recursion, we employ a conservative approach where the act of satisfying preconditions is terminated after the first recursive level. Algorithm 3 provides a high-level overview of the outlined search procedure. Assuming that there are T leaf actions to be scheduled and at most N potential slots for each such task, then the worst-case complexity of the algorithm is $O(TN)$ which is going to be heavily dominated by the size of slots since as each leaf gets scheduled $N \gg T$.

To summarize, our developed framework, `T-HTN` combines Algorithms 1, 2 and 3 to satisfy any incoming request given a set of timelines tied to an underlying STN. It first parses the incoming request and generates a corresponding path decomposition tree that gets processed by Algorithm 1 to generate all possible alternative decompositions. Each decomposition is then passed to Algorithm 2 to form a corresponding task network that enforces all the relevant temporal constraints in the underlying STN. The generated task network is then passed to Algorithm 3, which finds a set of feasible slots on the required resource timelines while attempting to satisfy any failing precondition literals. Since Algorithms 2 and 3 are repeated for all possible decompositions, the overall complexity of `T-HTN` also depends on the total number of such possible decompositions as defined by the input HTN. Assuming that there are at most D such decompositions, the worst-case complexity of our approach is $O(DTN)$ which is heavily dominated by Algorithm 3.

Looking back at the working example, assuming the robot parameter assignment in the corresponding task network shown in Figure 3 was UR5A, it is clear to observe that UR5B must move out of the way for the actions to take place successfully. This means that when `T-HTN` tries to schedule the instantiated task network with UR5A as the robot parameter assignment, a satisfying precondition procedure is triggered that calls the `m_clear_and_move` functional method. This method is internally linked to the specialized MAPF solver, which computes the best joint rail moves for both robots. Moreover, to pick the box, UR5A is first expected to be close to the object before attempting the grasp. This prerequisite condition also fails, triggering another `reachable` functional method that is also tied to

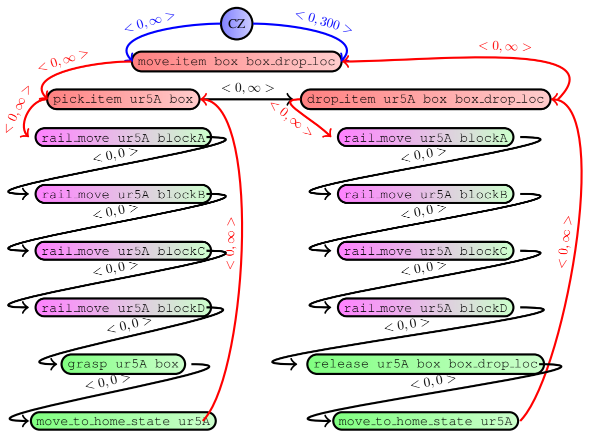

Figure 4: Updated instantiated task network for the working example as described in Figure 1. Nodes in red represent the high-level tasks, and the nodes in green represent the action primitives. The nodes in magenta were added to the original task network by a specialized planner who was triggered by `T-HTN` to satisfy a failing precondition. The blue edges correspond to the release time and due date constraints, while the red edges correspond to the contains constraint. The black edges correspond to the pre-specified synchronization rules.

the MAPF solver. In this case the path of the calling robot to the required destination is computed and installed in the plan. This results in the generation of an updated task network, which is shown in Figure 4. Figure 5 provides a final snapshot of the timelines generated by `T-HTN` as a result of scheduling the problem scenario outlined in the working example.

Continual Multiagent/Multi-Robot Planning

The core search procedure just summarized is repeatedly applied to incrementally generate, extend and manage multiagent/multi-robot plans over time as new pending requests and unexpected execution results that require replanning are received. New tasks are allocated to specific resources and integrated into the overall plan as new requests are received. In some cases, the remaining temporal flexibility in the current plan/schedule (or equivalently the continuing availability of required resources) will seamlessly accommodate additional requests. In other, more resource-constrained situations, the addition of new tasks may result in the delay or removal of some less important, previously scheduled tasks. In re-planning settings, it may also be the case that some previously planned/scheduled actions may no longer be relevant and can be retracted to create resource availability for performing corrective tasks. This inherently incremental search approach to planning and scheduling is well suited to such continual planning problems.

Experiments

To benchmark `T-HTN`'s performance, we designed a multi-request variant of the original scenario, which involves mov-

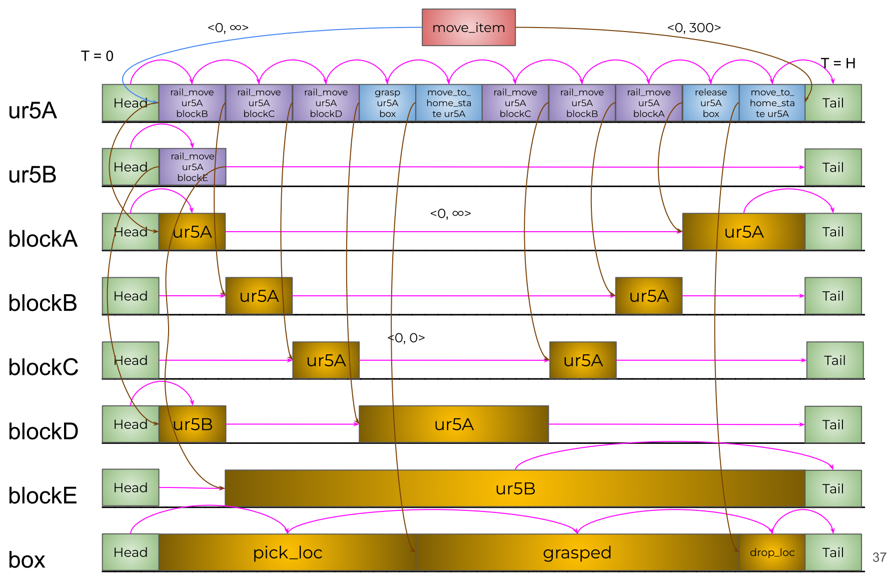

Figure 5: Final snapshot of the timelines generated by `T-HTN` in response to scheduling the working example outlined in Figure 1. The red node encapsulates the task network, which was shown in Figure 4 and the blue arrows signify the release time and due date constraints. The pink arrows relate to the $\langle 0, \infty \rangle$ sequencing constraint. In contrast, the brown arrows mark the $\langle 0, 0 \rangle$ contains constraint that joins the tokens on the *robot* and corresponding dependent timelines. The head and tail tokens on all the timelines act as auxiliary tokens, which do not have any significance apart from helping in a coherent token insertion procedure.

ing random objects from one location to another in the presence of a rail network that acts as a shared global resource constraint between the two robotic manipulators. Each request specifies the movement of a distinct, unique object from its initial location to another pre-specified destination location. All requests were given the same release-time and deadline constraints to facilitate comparative analysis. For the experiments, `T-HTN` utilizes an objective metric to prioritize plans that minimize makespan. The entire `T-HTN` planner, including the domain representation parser, was built in C++.

To evaluate the potential of the `T-HTN` framework, we compare its performance to another state-of-the-art planner that can be configured to optimize for makespan called POPF (Coles et al. 2010). POPF is a forward-chaining temporal planner built on the foundations of grounded forward search in combination with linear programming to handle continuous linear numeric change. Within the POPF domain specification, we specify one global deadline to be enforced on all requests, which is consistent with the common release and due dates specified in `T-HTN` input requests. We compare the two planners based on two metrics: computational cost to generate the plan and the resulting plan makespan. The experiments vary in the number of rail blocks, increasing the number of resources that must be managed, and the number of requests, which increases the size of the overall plan. We consider an experimental design that varies both the number of rail blocks and the number of requests from 5 to 25 in increments of 5. A time limit of 10 minutes was imposed for solution of any problem instance. All experiments were run on a Dell machine with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz to ensure a fair comparison.

Tables 1 and 2 show the results obtained relative to both performance objectives.With respect to makespan (Table 1), it can be seen that `T-HTN` outperforms POPF on the majority of problem instances solved by both techniques.

The more significant results are shown in Table 2. `T-HTN` dominates with respect to computational cost across all experiments and the differential increases significantly as problem size increases. Notably, POPF increasingly times out before generating a solution as the size of the problem grows. A natural question to ask with respect to computational cost is what impact the specialized MAPF algorithm had on comparative computational cost. To provide some insight, we conducted additional comparative experiments on problems involving just a single request that required multiple rail block moves, and in these experiments, it was found that `T-HTN` and POPF produced comparable compute times. Their average compute times over 10 different problem instances of single requests were 0.073 and 0.005 seconds respectively. Hence, it appears that the combinatorics of ordering multiple task requests dominates the computational cost of POPF solutions.

Summary

In this paper, we have presented a multiagent / multi-robot planning framework that combines the structural advantages of an HTN representation with the expressiveness and flexibility of timeline-based planning frameworks. We have argued that by emphasizing resource allocation as the basic decision-making focus, it is possible to overcome the complexity of the resulting expanded search space and efficiently

Requests	**5**		**10**		**15**		**20**		**25**	
Blocks	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**
5	620	**600**	1320	**1040**	1960	**1500**	2400	**2000**	3320	**2580**
10	**800**	**800**	2480	**1340**	Timeout	**2120**	Timeout	**2300**	Timeout	**3000**
15	1220	**1020**	**1900**	2060	Timeout	**2320**	Timeout	**3620**	Timeout	**5040**
20	**960**	1620	3600	**2560**	Timeout	**3320**	Timeout	**4620**	Timeout	**5780**
25	Timeout	**1320**	2280	**2200**	Timeout	**3620**	Timeout	**5020**	Timeout	**5260**

Table 1: Comparison of POPF and T-HTN with respect to the makespan of the generated plan.

Requests	**5**		**10**		**15**		**20**		**25**	
Blocks	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**	**POPF**	**T-HTN**
5	**0.22**	0.23	3.3	**0.54**	15.14	**2.03**	51.32	**3.43**	196.26	**10.61**
10	100.72	**0.37**	255.90	**0.87**	Timeout	**2.99**	Timeout	**5.63**	Timeout	**5.78**
15	2.64	**0.50**	94.00	**1.62**	Timeout	**3.61**	Timeout	**7.42**	Timeout	**11.56**
20	0.92	**0.91**	327.62	**2.84**	Timeout	**4.43**	Timeout	**17.29**	Timeout	**15.93**
25	Timeout	**0.76**	73.69	**2.12**	Timeout	**6.10**	Timeout	**10.88**	Timeout	**14.28**

Table 2: Comparison of POPF and T-HTN with respect to their computational times in seconds for generating a valid plan.

produce high quality multiagent plans.

To demonstrate this claim, we have developed the T-HTN planner/scheduler. Starting with the HDDL domain representation language, we introduced extensions to give resources and resource timelines special status, to include durative actions and incorporate complex temporal constraints between them, and to enable the use of specialized algorithms to solve well understood planning sub-problems. We then presented a core search algorithm that exploits these representational extensions to generate multiagent plans efficiently. Initial comparative experiments carried out in multi-robot scenarios involving two UR5 robot arms mounted on a shared rail network provided evidence in support of our overall design hypothesis.

Future Work

One immediate direction for future research is more extensive experimentation and analysis of T-HTN's performance characteristics. The results we have presented are preliminary and restricted to relatively simple sets of two-robot, object movement scenarios. We would like to expand experimentation to include other International Planning Competition (IPC) domain problems of general interest to the planning community. One complication here is mapping these domains and problems into the T-HTN's extended HDDL representation.

With regard to further development of T-HTN, a number of simplifying assumptions were made in its initial implementation that provide focal points for future research. First, resource timelines have been realized exclusively as single capacity resources (i.e., resources capable of doing just one task at a time). Although it appears straightforward, one short-term extension will be to extend resource timeline representations to accommodate multi-capacity robotic systems that can simultaneously accomplish multiple tasks (e.g., perform a visual inspection while carrying out a move-object request).

A second related simplification made in T-HTN was to restrict any agent (resource) from interleaving the execution of multiple task requests. Although interleaved accomplishment of multiple requests could lead to more efficient overall behavior in some circumstances, it also runs the risk of search space explosion. How to selectively relax this assumption is a longer term resource challenge.

Acknowledgements: This research was funded in part by the NASA Space Technology Research Institute for Deep Space Habitat Design under grant #80NSSC19K1052, and the CMU Robotics Institute.

References

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: A Constraint-based Planner for Generative and Hierarchical Temporal Planning. *ARXiv Preprint*.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'10, 42–49. AAAI Press.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence*, 49(1): 61–95.

Do, M.; and Kamhbampati, S. 2003. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 20: 155–194.

Eyerich, P.; Matmuller, R.; and Roger, G. 2009. Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, 130–137. AAAI Press.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences*, 18(2): 231–271.

Holler, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proceedings AAAI 2020*, 9883–9891. New York, NY.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. 1998. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2): 5–47.

Muscettola, N.; Smith, S.; Cesta, A.; and D'Aloisi, D. 1992. Coordinating Space Telescope Operations in an Integrated Planning and Scheduling Framework. *IEEE Control Systems*, 12(1).

Qi, C.; Wang, D.; Muoz-Avila, H.; Zhao, P.; and Wang, H. 2017. Hierarchical Task Network Planning with Resources and Temporal Constraints. *Knowledge-Based Systems*, 133(C): 17–32.

Rubinstein, Z.; Smith, S.; and Barbulescu, L. 2012. Incremental Management of Oversubscribed Vehicle Schedules in Dynamic Dial-A-Ride Problems. In *Proceedings AAAI 2012*. Toronto,.

Smith, D. E.; Frank, J.; and Cushing, W. 2007. The ANML Language.

Smith, S.; Becker, M.; and Kramer, L. 2004. Continuous Management of Airlift and Tanker Resources: A Constraint-Based Approach. *Mathematical and Computer Modeling*, 39(6-8): 581–598.

Umbrico, A.; Cesta, A.; Orlandini, A.; and Mayer, M. 2017. PLATINUm: A New Framework for Planning and Acting. In *16th International Conference of the Italian Association for Artificial Intelligence*.

Verfaillie, C., G.and Pralet; and Lematre, M. 2003. How to model planning and scheduling problems using constraint networks on timelines. *Knowledge Engineering Review*, 25(3): 319–336.