STAIRS 2014
U. Endriss and J. Leite (Eds.)

doi:10.3233/978-1-61499-421-3-51

HiPOP: Hierarchical Partial-Order Planning

Patrick BECHON , Magali BARBIER , Guillaume INFANTES , Charles LESIRE and Vincent VIDAL
Onera — The French Aerospace Lab; F-31055, Toulouse, France;
name.surname@onera.fr

Abstract. This paper describes a new planner, HiPOP (Hierarchical Partial-Order Planner), which is domain-configurable and uses POP techniques to create hierarchical time-flexible plans. HiPOP takes as inputs a description of a domain, a problem, and some optional user-defined search-control knowledge. This additional knowledge takes the form of a set of abstract actions with optional methods to achieve them. HiPOP uses this knowledge to enrich the output by providing a hierarchical time-flexible partial-order plan that follows the given methods. We show in this paper how to use this additional knowledge in a POP algorithm and provide results on a domain with a strong hierarchy of actions. We compare our approach with other temporal planners on this domain.

1. Introduction

The main focus of our approach is to deal with multi-agent missions where several teams of robots must collaborate and schedule concurrent tasks to achieve a common goal. In some cases, for instance for sea rescue [1], it is also compulsory to follow known patterns. This is especially appropriate when the system has to interact with humans trained to follow certain procedures. In this paper, we are only concerned with the initial plan production but our approach is designed to be easily used in cases where there is a need to execute and repair the plan.

In order to achieve those goals, we designed a planner that will output a time-flexible plan with hierarchical actions. A time-flexible plan will be easier to execute and to repair since small delays in actions can be dealt with without replanning everything. The hierarchical structure of the actions will allow the user to provide additional knowledge to the planner to improve the planning time and to impose additional constraints such as following some procedures.

We can use the hierarchical actions to plan on a higher level, for instance with teams or robots. And then use this plan at the team level to instantiate it into a plan for every robot. This leads to solutions where robots on the same team share the same high-level goal at any time (which is not expressible in term of low-level action) and they move together as much as possible. For a given high-level goal we can also describe how to achieve it with elementary actions. This also allows

the output of a higher-level solution to a human operator while each robot has the full plan with all elementary actions.

These reasons lead us to investigate in this first step both partial-order planning and hierarchical task network planning, aiming at obtaining a new algorithm that will mix the best of both worlds. The unification of those two approaches has already been discussed under the term *hybrid planning*.

The goal of HiPOP is to output a partial-order plan, with time flexibility and time concurrent actions, including a hierarchy among actions. The additional user-provided knowledge is also used to improve the planner performance. This knowledge being optional, HiPOP will resort to a POP algorithm if none is provided.

2. Background and related work

POP (Partial-Order Planning) is an algorithm already used by several planners such as VHPOP [20] and CPT [19]. One of the main drawbacks of POP compared to other commonly used algorithms is that it is usually slower [20]. But partial-order plans are more convenient to use in plan reparation [10], plan merging [8] and plan adaptation [11]. They can also output time-flexible plans.

On the other hand, introducing higher-level (abstract) actions like in Hierarchical Task Network (HTN) planning has been shown to improve planning time in some cases [13] and to help during plan reparation [6]. It is also more expressive than first principle planners [5]. Since the first HTN planners were state-based, they usually could not deal with time constraints and with concurrent actions. Some formalisms have been proposed to deal with those conditions, such as [2,7,13].

The idea of HiPOP to mix those two approaches was inspired by DPOCL [21], which introduced decomposition of actions in the context of POP. The same idea was already studied under the name *hybrid planning*, including the extension of hierarchical planning in UCP [9] and the PANDA system [16]. PANDA is a formal framework meant to compare different algorithms and heuristics, but not meant to be used in a real world setting.

This work can also be related to several ideas used in other situations. Adding preconditions and effects to higher-level actions to better control search is explored in Angelic planning [12]. GoDeL [17] uses optional user-defined knowledge to guide a landmark-based planner. Mixing HTN and POP in a domain-specific planner with a strict separation between several hierarchical levels is described in [3]. SIADEX [2] adds temporal reasoning into a HTN planner, keeping a forward-chaining algorithm.

HiPOP uses its POP component to output a time-flexible plan with concurrent actions, unlike state-based planners. It is able to plan with several levels of abstraction concurrently. And unlike other work on *hybrid planning*, HiPOP is implemented and compared to other temporal planners.

As HiPOP is an extension to classical POP, we first explain POP algorithm before introducing our additions. We then describe the search-control heuristics used, before explaining our experimental setup and showing some results.

3. Classical POP algorithm

We describe here a plain fully instantiated Partial-Order Planning algorithm (also called Partial-Order Causal Link). We selected a literal-based description of the world where a state is represented as a set of positive literals. The negation of literal l is noted $\neg l$. A domain is defined as a set of actions. The resulting state of the application of an action is obtained from a starting state by removing the set of its negative effects and adding the positive ones.

Definition 1 (action). *An action is a tuple* $(\mathcal{P}, \mathcal{E}, dur)$ *where* $\mathcal{P}$ *is a set of literals representing the preconditions,* $\mathcal{E}$ *is the set of literals representing the effects and* dur *is the duration of the action.*

Definition 2 (step). *A step* s *is a tuple* $s = (a, t_s)$ *where* $a = (\mathcal{P}, \mathcal{E}, dur)$ *is an action and* t_s *is an index of a time point in an STN. This timepoint represents the start time of* s*. We denote* $act(s) = a$*,* $t_{start}(s) = t_s$*,* $t_{end}(s) = t_s + dur$*,* $\mathcal{E}(s) = \mathcal{E}$*,* $\mathcal{P}(s) = \mathcal{P}$ *and* $dur(s) = dur$*.*

$t_{end}(s)$ represents the end time of the step s. It is used as a convenience notation to represent time constraints that only deal with the start time of each step. A Simple Temporal Network (STN) [4] is used to check schedulability over the time point indexes. If the set of constraints allows at least one solution, the STN (or equivalently the set of constraints) is said to be *consistent*.

Let s_i, s_j be steps. $s_i \prec s_j$ is a shorthand for $t_{end}(s_i) \leq t_{start}(s_j)$. We will use the classical definitions of causal links (noted $(s_i \xrightarrow{l} s_j)$ where l is a literal), open links (noted $(\xrightarrow{l} s_i)$) and threats (noted as a tuple $(s_k, s_i \xrightarrow{l} s_j)$ where s_k is the threatening step and $s_i \xrightarrow{l} s_j$ is the threatened causal link).

Definition 3 (flaw). *A flaw is either an open link or a threat.*

Definition 4 (partial plan). *A partial plan* P *is a tuple* $(\mathcal{S}, \mathcal{TC}, \mathcal{CL}, \mathcal{F})$ *where* $\mathcal{S}$ *is a set of steps,* $\mathcal{TC}$ *is a set of (simple temporal) constraints over the time points of* $\mathcal{S}$*,* $\mathcal{CL}$ *is a set of causal links,* $\mathcal{F}$ *is a set of flaws. We denote* $\mathcal{S}(P) = \mathcal{S}$ *and* $\mathcal{F}(P) = \mathcal{F}$*.*

A POP algorithm will explore the space of partial plans to find a complete plan. P is said to be *consistent* if $\mathcal{TC}(P)$ is consistent. P is said to be *complete* if $\mathcal{F}(P) = \emptyset$ and P is consistent.

Definition 5 (planning problem). *A problem instance is a tuple* $(\mathcal{A}, I, G)$ *where* $\mathcal{A}$ *is the set of available actions,* I *is a set of literals representing the initial state,* G *is a set of literals representing the goal.*

Algorithm 1 shows the pseudocode for a POP algorithm solving a planning problem $Prb = \{\mathcal{A}, I, G\}$. It works by keeping a set of all the generated but not yet visited plans: Π.

The initial plan is built by the procedure *InitialPartialPlan*. It creates a plan with two dummy steps, corresponding to actions a_s and a_e. a_s is the dummy start action with $\mathcal{P}(a_s) = \emptyset$, $\mathcal{E}(a_s) = I$ and a_e is the dummy end action with

Algorithm 1: Basic POP algorithm

```
1  Π = {InitialPartialPlan(I, G)} ;
2  while Π ≠ ∅ do
3  |   P = PopBestPlan(Π) ;
4  |   if F(P) = ∅ then
5  |   |   return P ;
6  |   end
7  |   f = PopBestFlaw(F(P)) ;
8  |   Π = Π ∪ Resolvers(A, P, f) ;
9  end
10 return ∅
```

$\mathcal{P}(a_e) = G$, $\mathcal{E}(a_e) = \emptyset$. All other steps must appears after the dummy start step and the dummy end step.

The *PopBestPlan* procedure removes the best partial plan from Π according to a heuristic and returns it. *PopBestFlaw* does the same with the set of flaws. At each iteration the algorithm selects the next plan to expand (line 3). Then a flaw is chosen (line 7) and the successors of P are generated and added to Π (line 8). The *Resolvers* $(\mathcal{A}, P, f)$ procedure returns a set of partial plans, each consistent and solving f in P in a different way accordingly to the type of the flaw. The algorithm stops when a complete plan is found (line 5) or when Π is empty (line 10).

To remove an open link to s, the *Resolvers* procedure has to add a causal link from s_i. s_i can be an existing step in P or a newly introduced step built from an action of $\mathcal{A}$. When adding a new causal link new threats can appear. When adding a new step all of its preconditions must be added as open links.

To remove a threat $(s_k, s_i \xrightarrow{l} s_j)$, there are only two ways: either the constraint $s_k \prec s_i$ (*demotion*) or $s_j \prec s_k$ (*promotion*) has to be added to the STN.

4. Adding abstract actions to classical POP

The goal of HiPOP is to use higher-level actions during search. The planner should be able to use abstract steps as elementary steps and to refine them when needed into a set of steps, causal links and temporal constraints.

Definition 6 (abstract action)**.** *An abstract action, also called higher-level action, is a tuple* $(\mathcal{P}, \mathcal{E}, dur, \mathcal{M}, \mathcal{C})$*:*

- *the first three elements* $(\mathcal{P}, \mathcal{E}, dur)$ *are the same as in an elementary action (Definition 1),*
- $\mathcal{M}$ *is a set of partial plans (called methods), used to instantiate the action,*
- $\mathcal{C}$ *is a set of conflicts (see Definition 8 below).*

A step with an abstract action is called an abstract step. The method of an abstract action is a partial plan in itself. During search, abstract actions can be

used as any other actions. But a new type of flaw is introduced: the abstract flaw. It represents the fact that there is an abstract step in the plan.

Definition 7 (flaw in abstract POP)**.** *(Replaces Definition 3) A flaw is an open link, a threat or the presence of an abstract step in P (abstract flaw).*

The only way to solve this new flaw is to pick one of its methods and to introduce the partial plan representing it in the current plan. It means adding all the steps, causal links, temporal constraints and flaws of the method to the plan, along with the dummy actions (see below). When adding the steps, a new time point is created for each one. After that, the abstract step has no effect on search (it cannot be used as the origin of a new causal link), but the newly created (non-dummy) steps can be used normally. The hierarchy of steps (which steps were introduced as children of which other steps) is also kept and returned at the end of the algorithm. Algorithm 1 is still valid, but *Resolvers* have to be adapted to deal with this new type of flaw.

Dummy actions (and their corresponding steps) are introduced in every partial plan. They are slightly different in case of methods associated to the abstract action a_i: the dummy actions a_i^s and a_i^e are such that $\mathcal{P}(a_i^s) = \mathcal{E}(a_i^s) = \mathcal{P}(a_i)$ and $\mathcal{P}(a_i^e) = \mathcal{E}(a_i^e) = \mathcal{E}(a_i)$; their duration is null. The difference with the dummy actions introduced in the initial plan is that the initial action has now a set of preconditions and the end dummy action has a set of effects.

They are introduced to deal with the following case. Assume that the open link $\xrightarrow{l} s_i$ exists when s_i is instantiated, where s_i is a step using a_i. Assume also that l is used by several steps in s_i. Then each step of s_i can be linked to the dummy initial step s_i^s. This open link is then the only one needed to guarantee that all the requirements of the child actions are met. Without dummy actions, we would have an open link for each step using l, increasing the number of plans to explore to solve them. This also guarantees than the same provider of l will be used for all steps in s_i.

Allowed actions. Using HiPOP as presented above leads to very poor performance and the output does not always take advantage of the hierarchical description of actions. This is because we only increased the branching factor, but even if the abstract actions are efficient the algorithm will explore in parallel plans with and without abstract actions. Those branches will produce the same elementary plan but will be explored concurrently.

To solve this issue we used an idea from HTN planning: the user provides a set of highest-level actions, the only actions that the planner can use to add a step. All the other actions are only used when refining an existing step. So the only change is in the *Resolvers* procedure, when adding a new step the algorithm can only select an explicitly allowed action. Planners like TALPlan and TLPlan also use the additional knowledge to prune the search tree, and this is similar to the search of plan respecting the *user-intent* presented in [9].

Threats of abstract actions. Some issues arise that can over-constrain the problem if threat solving procedures are not adapted to deal with abstract steps. Let us consider a threat $(s_k, s_i \xrightarrow{l} s_j)$. If s_k is abstract and encompasses several steps, it might be enough to promote only the last one and not all of them. If s_i is abstract and encompasses several steps, it might be enough to demote s_k before

the last step and not before all of them. If s_j is abstract and encompasses several steps, it might be enough to promote s_k after the first step and not after all of them. To deal with this problem, we use a new kind of promotion. The idea is to add only the mandatory constraint before the refinement even if it means that another constraint will need to be added after.

The constraint for demotion is $t^k_{dem} < t^i_{dem}$ and the constraint for promotion is $t^j_{pro} < t^k_{pro}$ where each variable, defined below, depends on the fact that the respective steps involved are elementary or abstract. If any step is abstract, the constraint will be loosened compared to the one enforced by the previous definition. If all the steps are elementary, the definitions are identical to the previous algorithm. It is generally not enough to ensure that the plan will be consistent. The algorithm will have to wait until the refinement to compute more precise threats.

- $t^i_{dem} \leftarrow t_{start}(s_i)$ **if** s_i is elementary **else** $t_{end}(s_i)$
- $t^k_{dem} \leftarrow t_{end}(s_k)$ **if** s_k is elementary **else** $t_{start}(s_k)$
- $t^j_{pro} \leftarrow t_{end}(s_j)$ **if** s_j is elementary **else** $t_{start}(s_j)$
- $t^k_{pro} \leftarrow t_{start}(s_k)$ **if** s_k is elementary **else** $t_{end}(s_k)$

If s_i is abstract we want to avoid scheduling s_k before the whole abstract step, so we restrict the demotion constraint to consider $t_{end}(s_i)$ instead of $t_{start}(s_i)$. If s_j is abstract we want to avoid scheduling s_k after the whole abstract step, so we restrict the promotion constraint to consider $t_{start}(s_j)$ instead of $t_{end}(s_j)$.If s_k is abstract we want to allow the promotion of only the last step of s_k (line 4) after s_j and the demotion of only the first step of s_k before s_i.

Another problem arises from the fact that a literal can be destroyed and recreated inside an abstract action. For instance in the *survivors* domain, one can create an abstract action where each team has a hospital as a homebase. Each step of this abstract action will move the team from its homebase to a survivor and back to the hospital. So the preconditions and the effects will have *(at ?team ?homebase)* but no causal link on the position of the team can be valid through the abstract step. It is inefficient to wait until the refinement of the abstract step to detect it.

To solve this issue, we introduced the notion of *resource conflicts*. Resource conflicts are provided in the description of abstract actions. They serve as a way to find which causal link does an abstract step threaten, independently of its effects. This also allows to detect two abstract steps that cannot appear concurrently.

Definition 8 (resource conflict)**.** *A resource conflict is a (hand-given) set of literals for an abstract action that may threaten causal links (but are not necessary a negative effect of the abstract action).*

Definition 9 (abstract resource conflict)**.** *Two abstract steps are in abstract resource conflict if the intersection of their resource conflicts is not empty.*

Abstract resource conflicts are considered as a new type of flaw. They can be solved in the same way than threats: removing a resource conflict between a_i and a_j if a_i and a_j are abstract is done by adding either $a_i \prec a_j$ or $a_j \prec a_i$ to the set of temporal constraints.

Soundness. The soundness proof of HiPOP follows the same pattern than the soundness proof of POP. If the algorithm returns a plan, one can remove all the abstract steps from this plan and only consider the elementary steps. The chain of causal links and the absence of threats are a guarantee that the plan is executable and accomplishes the goal.
Completeness. It highly depends on the available description and hypothesis. For instance literals can be masked by the hierarchical description if we assume that the elementary actions are forbidden. The proof is quick and easy if we allow the algorithm to plan with abstract or elementary actions without restriction, but this does not represent the actual use of the algorithm. The search is complete among the space of all plans that can be represented using only allowed action at the higher level, ie. among the plans respecting the *user intend*.

5. Search control

Algorithm 1 uses two heuristics to control search. On line 3 a first one is used to choose the next plan that will be expanded, called the *plan heuristic*. On line 7 another one is used to choose the next flaw that will be solved in a given plan, called the *flaw heuristic*. Those two heuristics are highly critical for the efficiency of HiPOP.
Plan heuristics. HiPOP uses the A^* algorithm to sort the set Π of all plans generated but not yet explored. They are stored in increasing order of $f(P) = g(P) + h(P)$ where $g(P)$ is the "distance" from the start point and $h(P)$ is a heuristic estimation of the cost to reach a complete plan from P. In HiPOP $g(P)$ is an estimation of the number of elementary (non-dummy) steps in P. It is an estimation because we cannot be sure of the number of steps in an abstract step if several methods are available to refine it. In this case, the minimum is taken. It can thus be defined recursively as:

$$g(P) = \sum_{s \in \mathcal{S}(P)} \begin{cases} 1 & \text{if } s \text{ is elementary} \\ \min\limits_{m \in \mathcal{M}(act(s))} g(m) & \text{if } s \text{ is abstract} \end{cases} \quad (1)$$

The computation of $h(P)$ uses the h_{add} heuristic as described by VHPOP [20]. Due to the lack of space we cannot describe it here, but we used the sum of the *cost* of each open link to sort plan, breaking ties using the *effort*.

The h_{add} heuristic does not take into account action reuse, and VHPOP proposes a modification of h_{add} to partially take care of it, called *reuse*.
Flaw heuristics. Previous work on VHPOP [20] as well as our initial results show that solving threat first is usually a good heuristic, especially when choosing first the ones with the fewest available resolvers.

To choose between open link flaws and abstract flaws, it is possible to always pick first abstract flaws or to mix their resolution with open link flaw or to pick them last.

Picking them first means that there is almost no planning done with abstract steps, but rather than they are used as a template of what steps should be created

together. Our tests showed that this leads to poor results since we do not have the benefits of planning with abstract steps but have to refine a lot of them.

Mixing the resolution of open link and abstract flaws is not yet studied in HiPOP due to the lack of heuristics that would allow to compare them. Instead, we focused on heuristics where the abstract flaws are solved at the end to plan as long as possible with abstract steps. This means that the planner will first compute an abstract plan, a plan whose only flaws are abstract, before refining them. This allows the planner to deal with smaller plans during the search, thus reducing the planning time until an abstract plan is found. If the abstract description ensures that any valid abstract plan can be refined into a complete plan without much backtrack, refining this abstract plan can be done quickly. And it also allows to separate the search on each instantiation of an abstract action.

While refining abstract steps, it is possible that some literals get "hidden" by the description: they are created by the elementary steps but are not in the abstract description (for instance they can depend on the choice of which method to use for this step). If those literals are needed to finish the plan, this can be an issue for the planner. To avoid this, the abstract flaws can be refined in their chronological partial order. If no literal is hidden and if the description allows a backtrack-free refinement, then the solving order does not matter.

To sort the open links, we used the same heuristics as the one described by VHPOP namely MW-Loc. It ranks open link according to the *effort* of their literals for the most recently added step. This is a compromise between staying focused on the current sub-goal and solving the harder literals first.

All the following results uses MW-Loc. It means that plans are sorted with $A*$ using the remaining *cost* as a heuristic. If the *costs* are equal, the remaining *effort* is used as a tie-breaker. The flaws are selected first on their type: *threats* first, then *open links* then *abstract flaw*. *Threats* are sorted in LIFO order. The first *open link* to be chosen is the one coming from the most recently added step, with the *effort* of this open link used as a tie-breaker. *Abstract flaws* are solved in chronological order (this is only a partial order, if two steps can be scheduled together, the flaws are solved in LIFO order).

6. Experimentation and results

We implemented HiPOP in C++, using the IPPC algorithm [14] to incrementally solve STNs. The domains and problems are modeled in PDDL and the abstract actions definition in a PDDL-like language. We created a random generator of problems for the *survivors* domain and a description of hierarchical actions. It is possible to vary the number of teams, of hospitals, of survivors and the size of zones. The position of hospitals and survivors are randomly chosen.

The goal of the hierarchical actions is to allow HiPOP to plan first by reasoning with teams (instead of individual robots) and zones (instead of individual locations). Once a plan with only abstract flaws is encountered, every abstract action will be instantiated into a set of elementary actions. Those elementary actions will be concerned with individual robots.

The only actions the planner is allowed to insert to solve an open link are elementary moves of a robot or a team, a hierarchical action to use a team to

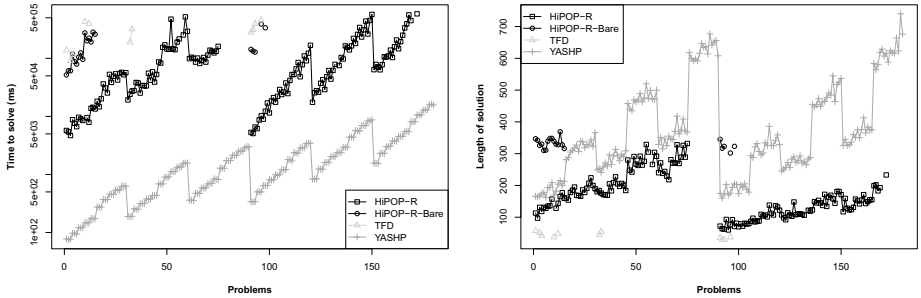

Figure 1. Time to solve (left) and makespan (right) on the *survivors* domain. HiPOP-R uses the *reuse* heuristics. HiPOP-R-Bare uses the same heuristics but without abstract actions. We also included the results of TFD (Temporal Fast Downward) and YASHP

explore a zone and a hierarchical action to use a team to bring a survivor back to a hospital. For each exploration action, there is one pre-computed patrol for a team to explore a zone that is not necessarily optimal. This means that the planner must still make sure that all actions are correctly chained up but does not need to solve a multi-vehicle travelling salesman problem on the whole zone. For each action bringing back a survivor, only a skeleton is given and the planner has to add some motion actions to create a valid plan.

We ran four different planners on a set of 180 randomly generated problems in the *survivors* domain. We used two versions of HiPOP, with the *reuse* heuristics : HiPOP-R and HiPOP-R-Bare. The latter does not use user-defined knowledge, so it is a classical POP algorithm. We also used two other temporal planners used in the temporal track of the IPC: Temporal Fast Downward [15] and YASHP [18]. The experiments were all run on an Intel X5670 processor running at 2.93Ghz with 24GB of RAM and a timeout of 10 min. The results are shown on Figure 1.

First we can see that adding the additional knowledge to HiPOP helps to reduce the planning time (by a factor of 10 on the first problems) and to improve the quality of the output plan. This mostly comes from the fact that HiPOP does not have to solve a generic travelling salesman problem but can use the precomputed patrol to solve it more quickly (even if it means that it cannot find the optimal solution, it will only find solutions that use those patrols).

Comparing it to other planners we can see that YASHP can very quickly find a solution that is almost always of poorer quality than that of HiPOP. On the opposite TFD is always slower than HiPOP but finds good solutions. The solutions found by TFD do not respect the abstract decomposition so they are not available to HiPOP.

7. Conclusion

We have shown how to add additional user knowledge into a POP planner and how this knowledge can be used by the planner to improve its performance. We presented HiPOP, a planning algorithm that uses this knowledge to output hierarchical, temporally flexible plans. The output plan also provides the information

about all the hierarchical actions that were used to generate the plan, so for each elementary action we can know the hierarchical action it belongs to.

For future work, we will further improve the performance of HiPOP. Several heuristics could be useful as shown by other research on POP algorithm, such as sorting the flaws by their number of resolvers. Another direction of study will also be to use the enriched plans to repair or merge them. It can also be used to better control its execution, in addition to the flexibility added by the least-commitment principle for temporal constraints used by POP.

References

[1] *International aeronautical and maritime search and rescue manual*, volume 3. IMO Publishing, 1998.

[2] L Castillo, J Fdez-Olivares, O Garcia-Pérez, and F Palao. Efficiently handling temporal knowledge in an HTN planner. In *ICAPS*, 2006.

[3] L Castillo, J Fdez-Olivares, and A Gonzalez. Integrating hierarchical and conditional planning techniques into a software design process for automated manufacturing. *Workshop on Planning under Uncertainty and Incomplete Information, ICAPS*, 2003.

[4] R Dechter, I Meiri, and J Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1):61–95, 1991.

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

[6] T Gateau, C Lesire, and M Barbier. HiDDeN: Cooperative Plan Execution and Repair for Heterogeneous Robots in Dynamic Environments. In *IROS*, Tokyo, Japan, 2013.

[7] RP Goldman. Durative Planning in HTNs. *ICAPS*, 2006.

[8] M A Hashmi and A El Fallah Seghrouchni. Merging of Temporal Plans Supported by Plan Repairing. *ICTAI*, 2010.

[9] S Kambhampati, A Mali, and B Srivastava. Hybrid planning for partially hierarchical domains. *AAAI/IAAI*, pages 882—-888, 1998.

[10] Roman Van Der Krogt and Mathijs De Weerdt. Plan Repair as an Extension of Planning. *ICAPS*, pages 161–170, 2005.

[11] S M Lee-Urban. *Hierarchical Planning Knowledge for Refining Partial-Order Plans*. PhD thesis, Lehigh University, 2012.

[12] B Marthi, S J Russell, and J Wolfe. Angelic semantics for high-level actions. In *ICAPS*, 2007.

[13] D Nau, T C Au, O Ilghami, U Kuter, J W Murdock, D Wu, and F Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404, 2003.

[14] L Planken, M de Weerdt, and N Yorke-Smith. Incrementally solving STNs by enforcing partial path consistency. *ICAPS*, 2010.

[15] G Röger, P Eyerich, and R Mattmüller. Tfd: A numeric temporal extension to fast downward. *6th IPC planners descriptions*, 2008.

[16] B Schattenberg. *Hybrid Planning And Scheduling*. PhD thesis, Ulm University, Institute of Artificial Intelligence, 2009.

[17] V Shivashankar, R Alford, U Kuter, and D Nau. The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning. *IJCAI*, 2013.

[18] V Vidal. YAHSP2: Keep it simple, stupid. *IPC*, pages 83–90, 2011.

[19] V Vidal and H Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.

[20] HLS Younes and RG Simmons. VHPOP: Versatile heuristic partial order planner. *Journal on Artificial Intelligence Research (JAIR)*, 20:405–430, 2003.

[21] RM Young, ME Pollack, and JD Moore. Decomposition and causality in partial-order planning. *International Conference on AI and Planning Systems (AIPS)*, 1994.