Handling Empty Decomposition Methods in Hierarchical Planning

Simona Ondrčková, Kristýna Pantůčková, Roman Barták
Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
{ondrckova, pantuckova, bartak}@ktiml.mff.cuni.cz

Abstract

Hierarchical planning is a form of planning where tasks decompose into sub-tasks until primitive tasks (actions) are obtained. These decompositions might contain additional constraints, such as subtask ordering and state constraints. If a task is already fulfilled, it does not need to decompose into anything, but it may still require satisfaction of a particular state constraint (to check that the task is fulfilled). Such decomposition methods are called empty. Despite practical usefulness, many hierarchical planning models do not support empty methods fully. This paper shows that two recently introduced hierarchical planning formalisms are equivalent with respect to empty methods. We also discuss the possibility of compiling such methods away. In particular, we show how to compile them away in totally ordered domains and discuss the difficulties in partially ordered domains.

Introduction

Planning is a technique that selects and organises actions into a sequence – a plan – to achieve a specific goal. In classical planning, this goal is described in a form of conditions that must be true after the sequence is finished. Each action may have effects that affect the state of the world and preconditions that must be true in order for the action to be executable. *Hierarchical planning* provides additional guidelines how to achieve particular tasks through decomposition into subtasks until actions are obtained (Ghallab, Nau, and Traverso 2004). Each of these decompositions might have additional constraints. A goal is typically described in the form of a goal (root) task that the planner must decompose.

Hierarchical planning has a variety of uses for example in automated assistance (Bercher et al. 2021), planning for spacecraft (Estlin, Chien, and Wang 1997) or machine learning (Mohr, Wever, and Hüllermeier 2018). Hierarchical plan verification is an opposite process to hierarchical planning. Given a plan and a goal task, the problem is to verify that the goal task correctly decomposes into the plan. This is useful to check that the plan complies with the hierarchical model.

To describe that some task is already achieved at a given state, the decomposition method may not contain any subtasks. Nevertheless, the method typically contains state constraints verifying that the task has been achieved in a given state. We call these tasks *empty tasks* and the decompositions *empty methods* and they will be the focus of this paper.

Many hierarchical planning formalisms do not deal with empty methods correctly (Ondrčková and Barták 2023). The traditional wisdom is that empty methods can be compiled away (Höller et al. 2014), but this does not assume method constraints. Recently, two new formalisms have been proposed to deal specifically with empty methods (Ondrčková and Barták 2023). One formalism, called a *No-op* model, handles empty methods by using *no-op()* actions – actions that do nothing. However, this does not result in the same plan as the plan is extended by the *no-op()* actions, which causes difficulties in plan verification. Therefore another formalism, called an *Index-Based* model, has been proposed to handle empty methods without creating extra actions. The transformation between plans of these formalisms to demonstrate their equivalence has not been shown yet. In this paper, we shall present such a transformation and show how the two models differ in regards to empty methods. We will also discuss how to compile the empty methods away. We will present this compilation for totally ordered domains and we will provide examples showing why the transformation is significantly more difficult in partially ordered domains.

Background

For actions in hierarchical planning (Erol, Hendler, and Nau 1996; Bercher, Alford, and Höller 2019) we use the STRIPS model (Fikes and Nilsson 1971). A world state is modelled as a set of atomic propositions that are true in that state and every other proposition is false (closed world assumption). An action is a 4-tuple of positive and negative preconditions and effects ($\texttt{pre}^+(a)$, $\texttt{pre}^-(a)$, $\texttt{eff}^+(a)$, $\texttt{eff}^-(a)$)). Preconditions represent what must be true in a state for the action to be applicable to it ($\texttt{pre}^+(a) \subseteq s, \texttt{pre}^-(a) \cap s = \emptyset$). The effects show how an action affects the state ($s' = (s \setminus \texttt{eff}^-(a)) \cup \texttt{eff}^+(a)$). The main difference between hierarchical and classical planning is that hierarchical planning introduces compound tasks that can decompose into subtasks (other tasks or actions). There might be multiple ways to decompose a task, each of these is described through a decomposition method. Let us assume we have a task T that decomposes to sub-tasks $T_1, ..., T_k$ under the constraints C.

The decomposition method is $T \to T_1, ..., T_k\ \ [C]$. Let U, V be subsets of tasks from T_1,...,T_k or T in which case the set contains all sub-tasks. We will use constraints as presented in a book by Ghallab et. al. (2004):

- $T_i \prec T_j$: an ordering constraint means that task T_i is before task T_j. The ordering is explicit here, it does not matter in which order the sub-tasks appear in the method.
- *before*(p, U): a precondition constraint means that in every plan, the proposition p holds in the state right before the first action to which set U decomposes.
- *after*(p, U): a postcondition constraint means that in every plan, the proposition p holds in the state right after the last action to which set U decomposes.
- *between*(U, p, V): a prevailing constraint means that in every plan, the proposition p holds in all states lying between the last action to which set U decomposes and the first action to which set V decomposes.

Let us assume a task *Get-To(V,L)* representing how to get a vehicle V to a desired location L. A decomposition method into an action *drive* could look like this:

$$\textit{Get-To(V,L)} \to drive(V, L_0, L) \tag{1}$$

If more steps are needed to get to the desired location, we can use a recursive decomposition:

$$\textit{Get-To(V,L)} \to \textit{Get-To}(V, L_0), drive(V, L_0, L) \tag{2}$$

What if the vehicle is already in the desired location? Then one can create an empty method decomposing task *Get-To* into nothing (ε) and checking the location:

$$\textit{Get-To(V,L)} \to \varepsilon\ \ [\textit{before(at(V,L),Get-To(V,L))}] \tag{3}$$

Let us differentiate between an empty task and an empty method. An empty method (Equation 3) is a method that decomposes a task into no sub-tasks. It may have some additional constraints in the form of *before* or *after* conditions. It cannot have *precedence* and *between* constraints as it does not decompose into anything so we can only attach constraints to the entire task (*precedence* and *between* constraints need at least two tasks). Note that the task this method decomposes might have other decompositions (empty or non-empty). An empty task is a task that has only one decomposition method and that method is empty. So task *Get-to* is not an empty task but it has an empty method.

Each method can be totally or partially ordered. Totally ordered method is a method, where all subtasks are linearly ordered. *Totally ordered domain* is a domain where each method is totally ordered. In totally ordered domains, each task decomposes into a continuous sequence of sub-tasks or actions. If a domain is not totally ordered, then we call it a *partially ordered domain*. Partial ordering allows interleaving of tasks – one task can decompose into an action lying between actions of another task. See example in Figure 1.

Let us now formalise a hierarchical planning problem: *Given a description of tasks (and actions), their decompositions, initial state S, and goal task G, does an executable action sequence (plan) exist, such that G decomposes into it?* This plan is the output.

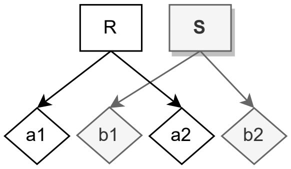

Figure 1: Task interleaving (actions of tasks R and S interleave in the plan).

Plan verification: *Given a description of tasks (and actions), their decompositions, initial state S, goal task G, and an action sequence (plan), can G be decomposed into the plan and is the plan executable?*

No-op Model

The *No-op* model transforms an empty method of task E into a regular method by decomposing the task E into a *no-op()* action – an action with no effects and no preconditions.

$$E \to \varepsilon\ \ [\textit{before(p,E)}] \tag{4}$$

All constraints that were part of the empty method (Equation 4) are moved to this new decomposition method:

$$E \to\ \ \textit{no-op()}\ [\textit{before(p,no-op())}] \tag{5}$$

The *no-op()* action marks the location in the plan, where the constraints of the empty method are checked. Detailed description of the model can be found in Ondrčková and Barták (2023). The disadvantage of this model is creating new actions in the plan that do not belong there. This is fine with planning (as they can be removed in the final plan) but it can be a problem with plan verification. As *no-op()* actions are not part of the plan to be verified, they need to be inserted to proper locations before the plan can be verified with respect to the *No-op* model. However, it is not clear in advance, where these locations should be.

Index-Based Model

The *Index-Based* Model deals with the problem of where the method constraints should be checked by using indexes. Each task T has two indexes, $start(T)$ and $end(T)$, representing the position (in the plan) of the first and the last action that the task decomposes to. For an empty task, both indexes point to a space, where the task lies in the plan. We call these half-indexes. For example, a task T that decomposes into the first and fifth action will have indexes: $start(T) = 1, end(T) = 5$. An empty task E that "decomposes" before the first action has indexes: $start(E) = 0.5$ and $end(E) = 0.5$. As the indexes are unknown until the plan is obtained, they are represented as variables in the style of constraint satisfaction. The method constraints can then be formulated as constraints over these variables:

- $T_i \prec T_j$: $\lfloor end(T_i) \rfloor < \lceil start(T_j) \rceil$,
- *before*(p, U): *before*$(p, \lceil start(U) \rceil)$,
- *after*(p, U): *after*$(p, \lfloor end(U) \rfloor)$,
- *between*(U, p, V): *between*$(\lfloor end(U) \rfloor$,p,$\lceil start(V) \rceil)$,

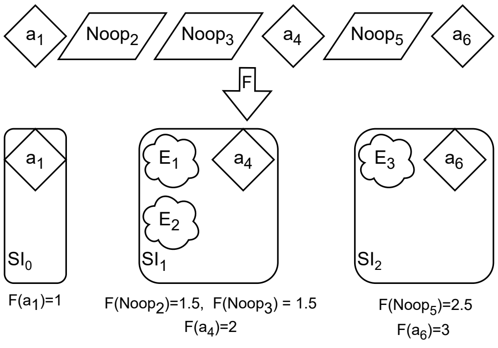

Figure 2: Mapping From No-op To Index Based Model

where $start(U) = min\{start(t')|t' \in U\}$, which is modelled using a new variable whose value equals the value of the formula. Similarly, $end(U) = max\{end(t')|t' \in U\}$.

The state constraints are then checked as follows:

- *before*(p, I): $p \in s_{I-1}$, *after*(p, I): $p \in s_I$,
- *between*(I, p, J): $\forall l, I \leq l < J : p \in s_l$.

For more detailed description check Ondrčková and Barták (2023). This model does not create any new actions so it can be used both in planning and in verification. In fact, one of the state of the art hierarchical plan verifiers uses the *Index-Based* model (Ondrčková et al. 2023).

Conversion Between Index and No-op Models

In this section we discuss the transformation between the *No-op* and *Index-Based* models. In particular, we will show that any plan that is valid in one model is also valid in the other model (after handling the *no-op* actions).

Transformation from *No-op* to *Index-Based* Model

Let us be given a plan – the *No-op* plan – consisting of actions $a_1, a_2, ...$ according to the *No-op* model. Some of the actions might be *no-op* actions. We will show, that if we remove the *no-op* actions from that plan, we will get a plan – the *Index-Based* plan – according to the *Index-Based* model. This is done by constructing function F mapping actions from the *No-op* plan to the *Index-Based* plan. Let $N(a_i)$ be the number of *no-op()* actions preceding the action a_i in the *No-op* plan. Then $F(a_i) = i - N(a_i)$ if a_i is a regular action. If a_i is a *no-op()* action then $F(a_i) = i - N(a_i) - 0.5$.

The *Index-Based* plan is obtained by removing all *no-op* actions from the *No-op* plan. The task decomposition trees for both plans are almost identical except the empty methods (see below). Let a_i be the first action and a_j be the last action to which some task T decomposes in the *No-op* model. We define the indexes for the task T in the *Index-Based* model as follows: $start(T) = F(a_i)$ and $end(T) = F(a_j)$. Notice that if T is an empty task, then $a_i = a_j =$ *no-op()* . In such a case, $start(T) = end(T)$ is pointing between two real actions in the *Index-Based* plan (half index).

Let us show an example plan in the *No-op* model (Figure 2): $a_1, noop_2, noop_3, a_4, noop_5, a_6$. *No-op* actions are obtained from tasks E using decomposition rules $E \rightarrow$ *no-op()* in the *No-op* model, while the *Index-Based* model uses empty methods there $E \rightarrow \varepsilon$. For example, if $noop_2$ is obtained from E_1 then $start(E_1) = end(E_1) = 1.5$. Notice that indexes of empty tasks in the *Index-Based* model point to proper locations in the plan so if there is any state constraint in the method, it will be checked at the correct state. There are seven states in the *No-op* plan, while the *Index-Based* plan contains just four states. However the states before and after *no-op()* actions are identical so no information is lost. For example S_1, S_2, and S_3 in the *No-op* plan are all identical and equal to state SI_1 in the *Index-Based* plan.

In order to easier refer to states in each model we will re-index the states in the *No-op* model so they better match the states of the *Index-Based* model: Let $Fa(a_i) = \lceil F(a_i) \rceil$. Each state before an action a_i will be indexed as $S_{(Fa(a_i)-1)_i}$. The state after last action l is numbered $S_{\lfloor F(a_l) \rfloor_l}$. This is because if the last action is a *no-op()* action then the state after it is the same as the state before it. So these are the states of our example: $S_{0_1}, S_{1_2}, S_{1_3}, S_{1_4}, S_{2_5}, S_{2_6}, S_{3_6}$. Note that $\forall x, y : S_{m_x} = S_{m_y}$. This is because every subsequent *no-op()* action and the first regular action after that share the same Fa. The states in the *Index-Based* model are called SI: $SI_0 = S_{0_1}$, $SI_1 = S_{1_2} = S_{1_3} = S_{1_4}$, $SI_2 = S_{2_5} = S_{2_6}$, $SI_3 = S_{3_6}$. Since *no-op()* actions and empty methods have no effect on the states we can see that $\forall i, x : SI_i = S_{i_x}$.

Next let us show that *before*, *after*, and *between* constraints are always checked at the same (equivalent) states. These conditions use sets of tasks U and V, but we already know which action is the first or last in these sets, so in the following proofs, we will use these actions.

Before Constraint: *Before* condition of an action a_i in the *No-op* model will be checked in the state $S_{(Fa(a_i)-1)_i}$. For the *Index-Based* model it will be checked at state: $SI_{\lceil start(a_i) \rceil - 1} = SI_{\lceil F(a_i) \rceil - 1} = S_{(Fa(a_i)-1)_i}$.

After Constraint: *After* condition of action a_i in the *No-op* model will be checked at state right before the following action a_{i+1}: $S_{(Fa(a_{i+1})-1)_{(i+1)}}$ or if no such action exists at final state $S_{\lfloor F(a_i) \rfloor_i}$ At the *Index-Based* model we should check the condition at $SI_{\lfloor end(a_i) \rfloor} = SI_{\lfloor F(a_i) \rfloor} = S_{\lfloor F(a_i) \rfloor_i}$. If a_i is not the last action, it remains to show that $S_{(Fa(a_{i+1})-1)_{(i+1)}} = S_{\lfloor F(a_i) \rfloor_i}$. For a regular action a_i, it holds $N(a_{i+1}) = N(a_i)$. If a_{i+1} is also a regular action, we get: $Fa(a_{i+1}) = \lceil F(a_{i+1}) \rceil = \lceil i + 1 - N(a_{i+1}) \rceil = \lceil i - N(a_i) + 1 \rceil = F(a_i) + 1$. If a_{i+1} is a *no-op()* action, we get: $Fa(a_{i+1}) = \lceil F(a_{i+1}) \rceil = \lceil i + 1 - N(a_{i+1}) - 0.5 \rceil = \lceil i - N(a_i) + 0.5 \rceil = F(a_i) + 1$. So $S_{(Fa(a_{i+1})-1)_{(i+1)}} = S_{(F(a_i)+1-1)_{(i+1)}} = S_{F(a_i)_{(i+1)}} = S_{F(a_i)_i} = S_{\lfloor F(a_i) \rfloor_i}$. If action a_i is a *no-op()* action, we can use the fact that the states before and after a *no-op()* action are the same. So $S_{(Fa(a_{i+1})-1)_{(i+1)}} = S_{(Fa(a_i)-1)_i} = S_{(\lceil F(a_i) \rceil - 1)_i} = S_{\lfloor F(a_i) \rfloor_i}$. The last step is due to the decimal part of $F(a_i)$ being equal to exactly 0.5 for *no-op()* actions.

Between Constraint: Let us assume a *between* condition that begins after action a_i and ends before action a_j. In the *No-op* model, the condition will be checked in states starting with $S_{(Fa(a_{i+1})-1)_{(i+1)}}$ or $S_{\lfloor F(a_i) \rfloor_i}$, if a_i is the last action (the state after action a_i) and ending with $S_{(Fa(a_j)-1)_j}$ (state before a_j). In the *Index-Based* model, we will check this condition between states $SI_{\lfloor end(a_i)\rfloor}$ and $SI_{\lceil start(a_j)\rceil-1}$. From previous, we already know $S_{(Fa(a_{i+1})-1)_{(i+1)}} = SI_{\lfloor end(a_i)\rfloor}$ and $S_{(Fa(a_j)-1)_j} = SI_{\lceil start(a_j)\rceil-1}$ so the same sets of states will be used.

Precedence Constraint: In the *No-op* model we check precedence constraints by checking that the last action of the first task is before the first action of the following task. Let us look at constraint $T_k \prec T_l$ and let us assume that the last action of T_k is a_i and the first action of T_l is a_j. Then we simply check that $i < j$. In the *Index-Based* model we use: $\lfloor end(T_k)\rfloor < \lceil start(T_l)\rceil$, that is, $\lfloor F(a_i)\rfloor < \lceil F(a_j)\rceil$. Independently of whether a_j is a real or *no-op()* action, we know $\lceil F(a_j)\rceil = j - N(a_j)$. Let a_i be a real action, then $\lfloor F(a_i)\rfloor = i - N(a_i)$. $i < j$ means that action a_i is before action a_j in the *No-op* plan so we can write $j = i + r + N(a_j) - N(a_i) + 1$, where $r \geq 0$ is a number of real actions between a_i and a_j. Hence, $i - N(a_i) = j - r - 1 - N(a_j) < j - N(a_j)$, which proves $\lfloor F(a_i)\rfloor < \lceil F(a_j)\rceil$. Let a_i be a *no-op()* action, then $\lfloor F(a_i)\rfloor = \lfloor i - N(a_i) - 0.5\rfloor = i - N(a_i) - 1$. Similarly to above, $i < j$ means that action a_i is before action a_j in the *No-op* plan, but because a_i is included in $N(a_j)$ we have $j = i + r + N(a_j) - N(a_i)$, where $r \geq 0$ is a number of real actions between a_i and a_j. Together, $i - N(a_i) - 1 = j - r - N(a_j) - 1 < j - N(a_j)$, which proves $\lfloor F(a_i)\rfloor < \lceil F(a_j)\rceil$.

Transformation from *Index-Based* to *No-op* Model

In the *Index-Based* plan, we have real actions (totally ordered) and empty tasks (lying between the real actions). The order is defined via the *start* indexes. It may happen that several empty tasks lie at the same location – they have identical *start* index (see Figure 2). These empty tasks still need to satisfy the ordering constraints, so we order them arbitrarily but according to these constraints. This gives a total order of real actions and empty tasks. We will now construct a function F' that maps actions and empty tasks in the *Index-Based* plan to positions in the *No-op* plan. This position indicates where the real action or a *no-op()* action (for an empty task) lies in the *No-op* plan. Let $N'(a)$ be the number of empty tasks before a in the *Index-Based* plan (using the ordering discussed above). Then $F'(a) = \lceil start(a)\rceil + N'(a)$. We shall show that F' is inverse to F by proving that $F(a_{F'(a)}) = start(a)$.

If a is a regular action then $F(a_{F'(a)}) = F'(a) - N(a_{F'(a)}) = (\lceil start(a)\rceil + N'(a)) - N(a_{F'(a)})$. Recall that $N(a_i)$ is the number of *no-op()* actions before action a_i in the *No-op* plan. For any empty task E lying before a in the *Index-Based* plan, it holds $start(E) < start(a)$ and $N'(E) < N'(a)$. Therefore, all such tasks E are mapped to *no-op()* actions before a in the *No-op* plan ($F'(E) < F'(a)$). On the other hand, no empty task E lying after a in the *Index-Based* plan, is mapped to *no-op()* action before a in the *No-op* plan ($F'(a) < F'(E)$). Therefore, $N(a_{F'(a)}) = N'(a)$ and so $F(a_{F'(a)}) = \lceil start(a)\rceil = start(a)$.

If a is an empty method then we get: $F(a_{F'(a)}) = F'(a) - N(a_{F'(a)}) - 0.5 = (\lceil start(a)\rceil + N'(a)) - N(a_{F'(a)}) - 0.5$. Using the same arguments as above we get $N(a_{F'(a)}) = N'(a)$ and hence $F(a_{F'(a)}) = \lceil start(a)\rceil - 0.5 = start(a)$ (recall that $start(a)$ is half-index).

We need to show now that the method constraints are satisfied in the *No-op* plan provided that they are satisfied in the *Index-Based* plan. Based on the notation introduced before, if a is a real action or empty task in the *Index-Based* plan, the state right before that real action or the *no-op()* action in the *No-op* plan is $S_{(\lceil start(a)\rceil-1)_{F'(a)}} = SI_{\lceil start(a)\rceil-1}$

Before Constraint: For an empty method or action a we check the condition at state: $SI_{\lceil start(a)\rceil-1}$. In the *No-op* model we will check it at state $S_{(Fa(a_{F'(a)})-1)_{F'(a)}} = S_{(\lceil F(a_{F'(a)})\rceil-1)_{F'(a)}} = S_{(\lceil start(a)\rceil-1)_{F'(a)}} = SI_{\lceil start(a)\rceil-1}$.

After Constraint: In the *Index-Based* plan we check the condition of action or empty task a at state $SI_{\lfloor end(a)\rfloor}$. In the *No-op* plan, this condition is checked in the state right before the action $a_{F'(a)+1}$, which is $S_{(Fa(a_{F'(a)+1})-1)_{(F'(a)+1)}}$. If a is the last action in the *Index-Based* plan then $SI_{\lfloor end(a)\rfloor}$ is the last state there. In such a case action $a_{F'(a)+1}$ does not exist in the *No-op* plan and the condition is checked in state $S_{\lfloor F(a_{F'(a)})\rfloor_{F'(a)}} = S_{\lfloor start(a)\rfloor_{F'(a)}} = SI_{\lfloor end(a)\rfloor}$.

Let us assume that b is an action or an empty task right after a in the *Index-Based* plan, then $SI_{\lfloor end(a)\rfloor} = SI_{\lceil start(b)\rceil-1}$ and $a_{F'(a)+1} = a_{F'(b)}$. Now we can write $S_{(Fa(a_{F'(a)+1})-1)_{(F'(a)+1)}} = S_{(Fa(a_{F'(b)})-1)_{F'(b)}} = SI_{\lceil start(b)\rceil-1} = SI_{\lfloor end(a)\rfloor}$.

Between Constraint: When the plan is given, checking the between constraint is equivalent to checking before constraints for specific actions and empty tasks, which we already proved to be equivalent for both models,

Precedence Constraint: If action or empty task a is before an action or empty task b in the *Index-Based* plan, then $F'(a) < F'(b)$. Hence any precedence constraint satisfied in the *Index-Based* plan is also satisfied in the *No-op* plan.

Compiling Away Empty Methods with State Constraints

One may ask, whether it is possible to compile empty tasks away from the hierarchical model, that is, to construct a hierarchical model generating the same set of plans but having no empty tasks. Höller et al. (2014) showed that this can be done for empty methods without state constraints. They used the model transformation known from context-free grammars that simply eliminates empty tasks by removing them from methods that decompose to them. However, this approach does not work when the empty method has a constraint attached to it. The reason is that we still need to check that constraint at some state. For example, in *Get-To(V,L)* task, we need to check that the vehicle V is at the destination location L (Equation 3).

Empty methods may contain only *before* and *after* constraints, but these constrains apply to the same state. For an empty task E with $start(E) = end(E) = i + 0.5$ we must check the after condition at $\lfloor end(E)\rfloor = i$ so at state s_i. A *before* condition of the same task must be checked at

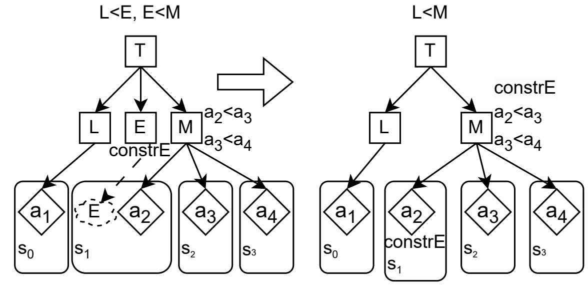

Figure 3: Passing Constraints From Empty Method to Following Task

$\lceil start(E) \rceil - 1 = i + 1 - 1$ so also at state s_i. Since both conditions are checked at the same state, we can transform any *after* constraint of an empty method into a *before* constraint. Hence without loss of generality we will assume only *before* constraints in empty methods.

Compiling Away Empty Methods In Totally Ordered Domains

Totally ordered domains have some specifics. For example, the *between* constraint can be compiled to *before* constraints and hence we may assume models without *between* constraints. This is done as follows, constraint *between*(U, p, V) in a method with subtasks *ST* is substituted by a set of constraints *before*(p, T) for each $T \in$ *ST* such that $\forall A \in U : A \prec T \wedge \neg\exists B \in V : B \prec T$. Similarly, if *before* and *after* constraints span over a set of subtasks, they can be converted to equivalent constraints using a single task. *before*(p, U) is substituted by *before*(p, T), where $T \in U$ such that $\neg\exists B \in U : B \prec T$ (such T is unique in totally ordered domains). *after*(p, U) is substituted by *after*(p, T), where $T \in U$ such that $\neg\exists B \in U : T \prec B$. This is an important observation as in totally ordered domains, we may assume only state constraints related to a single subtask.

In this section, we will show how to compile away empty methods with state constraints in totally ordered domains. The idea is moving these constraints to a task that directly follows the empty task in the plan (see Figure 3). If no such task exists, the *before* constraint is changed to *after* constraint and moved to a directly preceding task. If there is no preceding and no following task, we need to go one level up in the hierarchy. As tasks are totally ordered, one may locally find directly preceding and directly following task.

The first step in compiling empty methods away is to convert empty methods into empty tasks. Let us assume we have a task T with two decomposition methods: $M_1 : T \rightarrow A, B\ [C_1]$ and $M_2 : T \rightarrow \varepsilon\ [C_2]$. We remove method M_2, create a new task T', and add an empty method to it: $M' : T' \rightarrow \varepsilon\ [C_2]$. If there are multiple empty methods for one task we will create a new empty task for each method. If task T is a subtask in some method N, we add a new method N' where we substitute (some) T for T'. We do this for every empty method. We will also keep a record that will tell us which new tasks were created (T') and what task they were created from (T). We will call T the original task. Every time we create a new empty task from an empty method we check whether an empty task with the same conditions wasn't already created from the same original task. If yes, then we will not create it again. At the end of this process we have no empty methods that would not be related to an empty task. We create one empty task for each original empty method.

Next we will show how to remove an empty task. There are three options of how the method, where an empty task is a sub-task, can look:

Option A: The method contains a subtask that is right after the empty task E. For example $M_1 : T \rightarrow L, E, M\ [L \prec E, E \prec M]$. Since this is totally ordered, we know that task M decomposes into an action that immediately follows the empty task E (see Figure 3). If the empty task E contains constraint *before*(p, E) or such a constraint is part of M_1, we add this constraint to task M in the method. We transform method M_1 by removing task E from it and adding constraint *before* originally related to E and now related to M. This is how the new method looks $M'_1 : T \rightarrow L, M\ [L \prec M,$ *before*$(p, M)]$. Note that this transformation works even if M is an empty task as it will be removed later using a similar process. Also, if M_1 contains constraint *after*(p, E) or in general more state constraints related to E, they will all be added as *before* constraints related to M.

Option B: What if there is no task following an empty task in the decomposition, such as $M_1 : T \rightarrow L, E\ [L \prec E]$? Then we add the condition of the empty task E as an *after* condition of task L. If task E is at position $i + 0.5$, then its *before* constraint should be checked at state s_i. Since the domain is totally ordered we know that the last action, that L decomposes into, is at position a_i. We check the after condition for action a_i at state s_i, which is exactly what we need. We will transform method M_1, by removing task E from it and adding the *before* constraint of empty task E as an *after* condition on L. This is how the new method looks $M'_1 : T \rightarrow L\ [$*after*$(p, L)]$. Again, it works even if L is an empty task.

Option C: What if a task only decomposes into an empty task, such as $M_1 : T \rightarrow E\ [C]$? Then we remove E from the decomposition and we create a new empty method for task T. This means that it is possible to create a new empty method from an empty task. We get $M'_1 : T \rightarrow \varepsilon\ [C + C_E]$, where C_E are constraints of E (now applied to T). If there are no other decompositions of task T, then we simply get a new empty task T from empty task E and we continue trying to remove the next empty task. However, if there is another decomposition of T, then we need to use the first step of transforming an empty method into an empty task.

What about recursion? Since we transform an empty method into an empty task and then back, is not it possible to just keep creating new empty tasks infinitely? No, this is why we check when we create a new empty task from an empty method whether an empty task with the same conditions already exists and if so we will not create a new one.

Let us look at an example of $T \rightarrow A\ [C_T]$ and $A \rightarrow T\ [C_A]$, with an empty method $M : T \rightarrow \varepsilon\ [C_E]$. We first transform this empty method M into an empty task T' so we get: $T \rightarrow A\ [C_T]$, $A \rightarrow T'\ [C_A]$. Then we remove T' so we get a new empty method on A:$A \rightarrow \varepsilon[C_A, C_E]$.

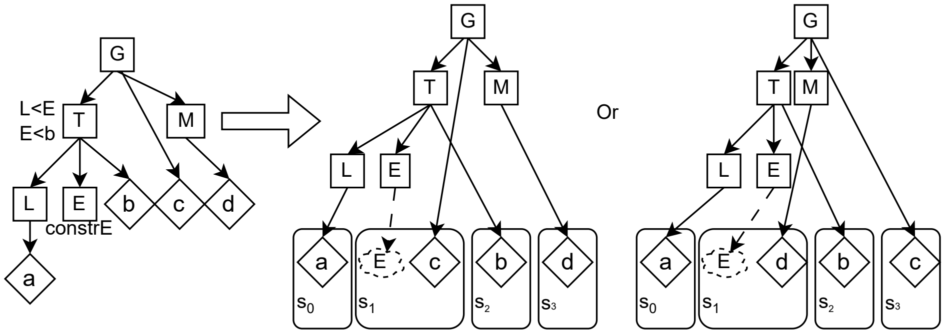

Figure 4: Passing Constraints From Empty Method to Following Task.

Note that this new empty method has both the original constraints of the decomposition of A and the constraints of the empty task E. We continue this by transforming it into an empty task A'. $T \rightarrow A'$ $[C_T]$, $A' \rightarrow \varepsilon$ $[C_A, C_E]$ Then we remove the empty task A': $T \rightarrow \varepsilon$ $[C_T, C_A, C_E]$. Now we must turn this empty method into an empty task, but we already have T' so we create $T'' \rightarrow \varepsilon$ $[C_T, C_A, C_E]$. We then once again remove it. So we must again create a new empty method for A: $A \rightarrow T''$, which then creates new empty task $A'' \rightarrow \varepsilon$ $[C_T, C_A, C_E]$ but the continual creation of new empty tasks stops once we remove the empty task A''. This is because we get a new empty method: $T \rightarrow \varepsilon$ $[C_T, C_A, C_E]$, we attempt to create a new empty task T''' but there already exists an empty task T'' with the same condition that was created from the same original task. The number of conditions in the domain is finite and therefore the algorithm will eventually stop.

Why Is It Hard to Compile Away Empty Methods In Partially Ordered Domains?

In order to compile empty methods away in totally ordered domains, we passed the constraints onto the next following task, which then checks the constraints at the state right before the following action. Could we do the same in partial ordering? Let us look at how would we find this next action. Partial ordering allows interleaving (see Figure 1). So a task that is in a different level of the decomposition and that does not relate to the empty method at all might decompose into the immediately following action. We can see two examples of this in Figure 4. In one case, the task G decomposes into the following action c and in the other case it is the task M that decomposes to the following action d. Notice that they both interleave with task T, the parent of the empty task E. These are just two examples of possible decompositions as there are no ordering constraints between tasks G, M and their sub-tasks.

We could attempt to get all possible orderings between actions and empty methods to deduce which actions can be after a specific empty method. However, in order to do this, we need to differentiate same actions that were create from different parent tasks. This requires us to calculate all the possible decompositions of the model. Another problem arises with recursion that may lead to infinite number of possible decompositions.

Let us assume for a moment that we can find the following action for an empty task and we can put the *before* condition of the empty method into it. There is still another problem we must solve and that is ordering. If an empty task is the first or last sub-task of an ordered task, then by removing the empty method we also remove the marker of where the other task should end/start. Let us show this on an example: $M \rightarrow O, P$ $[O \prec P]$; $P \rightarrow E, L[E \prec L]$; $E \rightarrow \epsilon[C]$. Let us imagine that $start(E) = 2.5$ and $start(L) = 5$ (this is possible due to interleaving so actions of other tasks may lie between E and L). Then $start(P) = start(E) = 2.5$. For the plan to be valid, task O must have $end(O) < \lceil 2.5 \rceil$. Let us now assume that we move the constraint from the empty method to the immediately following action and we also remove task E. So now task P has $start(P) = 5$ and therefore task O may have *start* index as high as 4.5 (as opposed to 2.5). This is clearly a relaxation of the original problem. What happens is that if an empty task is the first sub-task of some parent task P, then by removing the empty task we lose the starting point of that task P, which makes any precedence constraints related to task P invalid.

Conclusion

In this paper we focused on empty methods in hierarchical planning. We showed that two recently introduced models for hierarchical planning are equivalent in the sense that arbitrary plan in one model can be transformed to a plan in the other model. We also showed than in totally-ordered domains, it is possible to compile empty methods away even if they have constraints attached. For partially-ordered domains, we discussed some difficulties that appear when one attempts to compile empty methods away. This problem is still open and it poses a challenge for future research.

Acknowledgments

Research is supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215 and by SVV project number 260 698 and Simona Ondrčková is supported by the Charles University, project GA UK number 280122.

References

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – one abstract idea, many concrete realizations. In *IJCAI 2019*, 6267–6275. IJCAI.

Bercher, P.; Behnke, G.; Kraus, M.; Schiller, M.; Manstetten, D.; Dambier, M.; Dorna, M.; Minker, W.; Glimm, B.; and Biundo, S. 2021. Do it yourself, but not alone: *Companion*-technology for home improvement – bringing a planning-based interactive DIY assistant to life. *Künstliche Intelligenz – Special Issue on NLP and Semantics* 35:367–375.

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

Estlin, T. A.; Chien, S. A.; and Wang, X. 1997. An argument for a hybrid htn/operator-based approach to planning. In *Recent Advances in AI Planning: 4th European Conference on Planning, ECP'97 Toulouse, France, September 24–26, 1997 Proceedings 4*, 182–194. Springer.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *IJCAI 1971*, 608–620.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

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

Mohr, F.; Wever, M.; and Hüllermeier, E. 2018. ML-plan: Automated machine learning via hierarchical planning. *Machine Learning* 107(8):1495–1515.

Ondrčková, S., and Barták, R. 2023. On semantics of hierarchical planning domain models with decomposition constraints and empty methods. In *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, 349–353. IEEE.

Ondrčková, S.; Barták, R.; Bercher, P.; and Behnke, G. 2023. Lessons learned from the cyk algorithm for parsing-based verification of hierarchical plans. In *The International FLAIRS Conference Proceedings*, volume 36.