Handling Empty Decomposition Methods in Hierarchical Planning

Simona Ondrčková, Kristýna Pantůčková, Roman Bartáč
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 sub-tasks. 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 Bartáč 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 Bartáč 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 demonstrates 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 (pre+(a), pre−(a), eff+(a), eff−(a)). Preconditions represent what must be true in a state for the action to be applicable to it (pre+(a) ⊆ s, pre−(a) ∩ s = ∅). The effects show how an action affects the state (s' = (s\eff−(a)) ∪ eff+(a)). The main difference between hierarchical and classical planning is that hierarchical planning introduces compound tasks that can decompose into sub-tasks (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 T1, ..., Tk under the constraints C.
The decomposition method is \( T \rightarrow T_1, \ldots, T_k \ [C] \). Let \( U, V \) be subsets of tasks from \( T_1, \ldots, T_k \) 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.
- \( \text{before}(p, U) \): 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.
- \( \text{after}(p, U) \): 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.
- \( \text{between}(U, p, V) \): 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 \( \text{Get-To}(V, L) \) representing how to get a vehicle \( V \) to a desired location \( L \). A decomposition method into an action \( \text{drive} \) could look like this:

\[
\text{Get-To}(V, L) \rightarrow \text{drive}(V, L_0, L)
\]  

(1)

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

\[
\text{Get-To}(V, L) \rightarrow \text{Get-To}(V, L_0), \text{drive}(V, L_0, L)
\]  

(2)

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

\[
\text{Get-To}(V, L) \rightarrow \varepsilon \ [\text{before}(V, L), \text{Get-To}(V, L)]
\]  

(3)

Let us differentiate between an empty task and an empty method. An empty method (Equation 4) 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 \( \text{Get-to} \) 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. \( \text{Totally ordered domain} \) 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 \( \text{Partially ordered domain} \). 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).](image)

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.

\[
E \rightarrow \varepsilon \ [\text{before}(p,E)]
\]

(4)

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

\[
E \rightarrow \text{no-op()} \ [\text{before}(p,\text{no-op()} )]
\]

(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 Ondrˇkov´a and Bart´ak (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, \( \text{start}(T) \) and \( \text{end}(T) \), 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: \( \text{start}(T) = 1 \) and \( \text{end}(T) = 5 \). An empty task \( E \) that “decomposes” before the first action has indexes: \( \text{start}(E) = 0.5 \) and \( \text{end}(E) = 0.5 \). 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:

- \( T_i \prec T_j \): \( \lfloor \text{end}(T_i) \rfloor < \lfloor \text{start}(T_j) \rfloor \),
- \( \text{before}(p, U) \): \( \text{before}(p, \lfloor \text{start}(U) \rfloor) \),
- \( \text{after}(p, U) \): \( \text{after}(p, \lfloor \text{end}(U) \rfloor) \),
- \( \text{between}(U, p, V) \): \( \text{between}(\lfloor \text{end}(U) \rfloor, p, \lfloor \text{start}(V) \rfloor) \),
Let us assume a *between condition* that begins after action $a_i$ and ends before action $a_j$. If $a_i$ is the last action, we get: $F(a_{i+1}) = \lfloor F(a_i) + 1 \rfloor$ or $S(F(a_{i+1})) = \lfloor S(F(a_i)) + 1 \rfloor$. If $a_i$ is not the last action, it remains to show that $S(F(a_{i+1}))(i+1) = (i+1)$. 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: $F(a_{i+1}) = \lfloor F(a_{i+1}) + 1 \rfloor$ or $S(F(a_{i+1})) = \lfloor S(F(a_i)) + 1 \rfloor$. If $a_{i+1}$ is a no-op action, we get: $F(a_{i+1}) = \lfloor F(a_i) \rfloor + 1$ or $S(F(a_{i+1})) = \lfloor S(F(a_i)) \rfloor + 1$. 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 $S(F(a_{i+1})) = S(F(a_i)) + 1 = S(F(a_i))(i+1) = S(F(a_i))(i+1)$. 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 $S(F(a_{i+1})) = S(F(a_i))(i+1)$ or $S(F(a_i))(i+1) = S(F(a_i))(i+1)$. The last step is due to the decimal part of $F(a_i)$ being exactly 0.5 for no-op actions.

**Before Constraint:** Let us assume a *between condition* that begins before action $a_i$ and ends after action $a_j$. In the *Index-Based* model, the condition will be checked in states starting with $S(F(a_i)) = S(F(a_i))$ or $S(F(a_i)) = S(F(a_i))$. The last step is due to the decimal part of $F(a_i)$ being exactly 0.5 for no-op actions.

**After Constraint:** Let us assume a *between condition* that begins after action $a_i$ and ends before action $a_j$. In the *Index-Based* model, the condition will be checked in states starting with $S(F(a_{i+1})) = S(F(a_i))(i+1)$ or $S(F(a_i))(i+1) = S(F(a_i))(i+1)$. The last step is due to the decimal part of $F(a_i)$ being equal to exactly 0.5 for no-op actions.

**Before Constraint:** Let us assume a *between condition* that begins before action $a_i$ and ends after action $a_j$. In the *Index-Based* model, the condition will be checked in states starting with $S(F(a_{i+1})) = S(F(a_i))(i+1)$ or $S(F(a_i))(i+1) = S(F(a_i))(i+1)$. The last step is due to the decimal part of $F(a_i)$ being exactly 0.5 for no-op actions.

**After Constraint:** Let us assume a *between condition* that begins after action $a_i$ and ends before action $a_j$. In the *Index-Based* model, the condition will be checked in states starting with $S(F(a_{i+1})) = S(F(a_i))(i+1)$ or $S(F(a_i))(i+1) = S(F(a_i))(i+1)$. The last step is due to the decimal part of $F(a_i)$ being exactly 0.5 for no-op actions.
(the state after action \(a_i\)) and ending with \(S_{(Fa(a_i)-1)}\) (state before \(a_i\)). In the Index-Based model, we will check this condition between states \(SI_{[\text{end}(a_i)]}\) and \(SI_{[\text{start}(a_i)]-1}\). From previous, we already know \(S_{(Fa(a_i+1)-1)} = \sum_{i=1}^{j} N_{(a_i)}\). 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 \(T_k < T_l\) 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: \([\text{end}(T_k)] < [\text{start}(T_l)]\), that is, \([F(a_i)] < [F(a_j)]\). Independently of whether \(a_i\) is a real or no-op() action, we know \([F(a_i)] = i - N(a_i)\). Let \(a_i\) be a real action, then \([F(a_i)] = i - N(a_i)\) 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 \([F(a_i)] < [F(a_j)]\). Let \(a_i\) be a no-op() action, then \([F(a_i)] = i - N(a_i) - 0.5\) = \(i - N(a_i) - 1\). 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 \([F(a_i)] < [F(a_j)]\).

**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 \(\text{start}\) indexes. It may happen that several empty tasks lie at the same location—they have identical \(\text{start}\) 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 \(F'(a) = [\text{start}(a)] + N'(a)\). We shall show that \(F'\) is inverse to \(F\) by proving that \(F(a_{F'(a)}) = \text{start}(a)\).

If \(a\) is a regular action then \(F(a_{F'(a)}) = F(a) - N(a_{F'(a)}) = ([\text{start}(a)] + N'(a)) - N(a_{F'(a)})\). 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 \(\text{start}(E) < \text{start}(a)\) and \(N'(E) < N'(a)\). Therefore, all such tasks \(E\) are mapped to no-op() actions before \(a\) in the No-op plan \(F'(a) < F'(a)\). 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 \(F'(a) < F'(a)\). Therefore, \(N(a_{F'(a)}) = N'(a)\) and so \(F(a_{F'(a)}) = [\text{start}(a)] = \text{start}(a)\).

If \(a\) is an empty method then we get: \(F(a_{F'(a)}) = F(a) - N(a_{F'(a)}) - 0.5 = ([\text{start}(a)] + N'(a)) - N(a_{F'(a)}) - 0.5.\) Using the same arguments as above we get \(N(a_{F'(a)}) = N'(a)\) and hence \(F(a_{F'(a)}) = [\text{start}(a)] - 0.5\) (recall that \(\text{start}(a)\) 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. From 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 \(S_{[\text{start}(a)]-1} = SI_{[\text{start}(a)]-1}\).

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

**After Constraint:** In the Index-Based plan we check the condition of action or empty task \(a\) at state \(SI_{[\text{end}(a)]}\). In the No-op plan, this condition is checked in the state right before the action \(a_{F'(a)+1}\), which is \(S_{(Fa(a_{F'(a)})-1)} = S(a_{F'(a)})\). If \(a\) is the last action in the Index-Based plan then \(SI_{[\text{end}(a)]}\) 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 \(S_{(Fa(a_{F'(a)})+1)} = S_{[\text{start}(a)]+1} = SI_{[\text{end}(a)]}\).

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

**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 \(\text{GetTo}(V,L)\), 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 constraints apply to the same state. For an empty task \(E\) with \(\text{start}(E) = \text{end}(E) = i + 0.5\) we must check the after condition at \(\text{end}(E) = i\) so at state \(s_i\). A before condition of the same task must be checked at

\(F'(a) - N(a_{F'(a)}) - 0.5 = ([\text{start}(a)] + N'(a)) - N(a_{F'(a)}) - 0.5.\) Using the same arguments as above we get \(N(a_{F'(a)}) = N'(a)\) and hence \(F(a_{F'(a)}) = [\text{start}(a)] - 0.5\) (recall that \(\text{start}(a)\) is half-index).
After 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 ∈ ST such that ∀A ∈ U : A < T ∧ ¬∃B ∈ V : B < 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 ∈ U such that ¬∃B ∈ U : B < T (such T is unique in totally ordered domains). after(p, U) is substituted by after(p, T), where T ∈ U such that ¬∃B ∈ U : T < 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: M₁ : T → A, B [C₁] and M₂ : T → ε [C₂]. We remove method M₂, create a new task T', and add an empty method to it: M₃ : T' → ε [C₂]. 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 M₁ : T → L, E, M [L < E, E < 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 M₁, we add this constraint to task M in the method. We transform method M₁ 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 M₁′ : T → L, M [L < 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 M₁ 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 M₁ : T → L, E [L < 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 sᵢ. Since the domain is totally ordered we know that the last action, that L decomposes into, is at position aᵢ. We check the after condition for action aᵢ at state sᵢ, which is exactly what we need. We will transform method M₁, 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 M₁′ : T → 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 M₁ : T → 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 M₁′ : T → ε [C + C_E], where C_E is 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 → A [C_A] and A → T [C_T], with an empty method M : T → ε [C_E]. We first transform this empty method M into an empty task T' so we get: T → A [C_A], A → T' [C_A]. Then we remove T' so we get a new empty method on A : A → ε [C_A, C_E].
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 $A' \rightarrow T \rightarrow A' [C_T, A'] \rightarrow \varepsilon [C_A, C_E]$ Then we remove the empty task $A': T \rightarrow T \rightarrow \varepsilon [C_T, C_A, C_E]$. 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 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: $M \rightarrow O, P [O \prec P]; P \rightarrow \varepsilon, L[\varepsilon \prec L]; \varepsilon \rightarrow C$. 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 $end(O) < [2.5]$. 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 that 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 Ondrˇckov´a is supported by the Charles University, project GA UK number 280122.
References


