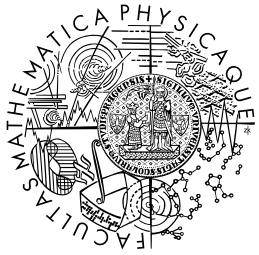

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Kroupa

Semantics and transformations of HTN models

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Prof. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I am deeply grateful to my supervisor, Prof. RNDr. Roman Barták, Ph.D., for proposing such an interesting thesis topic that perfectly suited my interests, and for his invaluable guidance and support.

I would also like to thank all the teachers who have taught me throughout my life, especially those at MFF UK.

Last but not least, I want to express my sincere gratitude to my parents and close friends for their unconditional support.

Title: Semantics and transformations of HTN models

Author: David Kroupa

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Hierarchical task network (HTN) is an approach to planning where compound tasks are decomposed into subtasks that can be either decomposed or executed right away. HTN is an extension of classical planning analogously as context-free languages extend regular languages. The goal of the thesis is split into two parts: describe, compare, and analyze various semantics and propose transformations of HTN models that do not lose any characteristics about the model. A significant piece of the thesis aims at handling empty methods which are usually not defined properly. The biggest challenges of HTN transformations lie in the proper management of constraints.

Keywords: hierarchical planning, hierarchical task networks, semantics, transformations

Název práce: Sémantiky a transformace HTN modelů

Autor: David Kroupa

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Hierarchická síť úloh (HTN) je způsob plánování obsahující složené úlohy, které lze dekomponovat na podúlohy, které mohou být opět dekomponovány nebo přímo vykonány. HTN je rozšíření klasického plánování podobně jako bezkontextové jazyky rozšiřují regulární jazyky. Tato práce má dva cíle. Prvním je popsat, porovnat a analyzovat různé sémantiky. Druhým je navrhnout transformace HTN modelů, které by neztrácely žádné informace a vlastnosti modelů. Značná část této práce se zabývá vyřešením problémů prázdných metod, které většinou nejsou moc dobře zpracovány. Největší výzva HTN transformací je správná správa plánovacích omezujících podmínek.

Klíčová slova: hierarchické plánování, hierarchické sítě úloh, sémantika, transformace

Contents

Introduction

Anyone with internet access must have noticed the rapid development of artificial intelligence (AI). What used to be science fiction in the past (not that many years ago) is now becoming more and more relevant. Some people are fascinated and others are terrified by the capabilities of modern AI technologies, ranging from large language models (LLM) to generative video makers with precise details.

We can have lengthy and deeply philosophical discussions on the definition and meaning of intelligence. Every living being has some kind of intelligence. Humans, animals, insects, and even trees have some level of intelligence (someone might use the word consciousness). We may not conclude on the final form of this definition, but surely we will all agree that a substantial part of intelligence is decision-making and planning. From the variety of all possible actions, we choose one that is the most suitable and optimal for the current situation. In situations where we want to attain a complex goal, we need to use our intelligence and create a plan, i.e., a sequence of valid actions that will change the current state of the world to the desired one.

This thesis concerns the Hierarchical Task Network (HTN) which is the way most people think about problems and ways of solving them. Having some abstract and high-level tasks, we may decompose these tasks into subtasks that may be later again decomposed or executed right away. A high-level task is accomplished when all of its subtasks are accomplished. HTN can express different planning domains more compactly and intuitively than classical planning, which will be discussed in this thesis as well.

Motivation

The theory of classical and hierarchical planning is closely related to the theory of Automata and Grammars [1] [2] [3] [4]. For this reason, we can utilize concepts, knowledge, and well-known structures from the theory of Automata and Grammars and apply them to the field of planning. Planning and especially hierarchical planning is widely used in AI and other similar fields of informatics.

If we want to improve AI, then we need to improve planning, which is part of the concept of intelligence, as we discussed earlier. If we want to improve some theoretical knowledge, then we need to do theoretical research in that field. Hence, this bachelor's thesis was created, to at least somehow help the international community of researchers in this particular field. It is important to remember the saying, "Rome was not built in a day".

Goal

The goal of this thesis is a consequence of what was said in the **Motivation** section. We want to explore the theory of planning by taking advantage of the existing knowledge of Automata and Grammars. The title of this thesis consists of words **Semantics** and **Transformations** thus the scope of exploration is set

so. One of the subgoals is to assemble valuable information about the theory of planning that is spread throughout different publications and textbooks. In the semantics part, we want to describe and compare different formalisms and their characteristics. Moreover, we will introduce new semantics regarding hierarchical planning. In the transformations part, we want to find ways of modifying the hierarchical models without losing any properties of the models. For example, we might try to compile away some constraints and convert them into different constraints.

Structure

In chapters one and two, we will describe and define classical planning and HTN planning, respectively. There are many different ways of representing both planning models, however, we will outline only the most known representations. In the third chapter, we will characterize various semantics of HTN planning. That is, what is the meaning of HTN models, and how we should think about them. Finally, the fourth chapter shows some transformations, that might help achieve prerequisites for particular HTN planners, for example, to compile away unsupported features.

1 Classical Planning

Before we dive into HTN planning, it's essential to understand its predecessor, which is classical planning or, as it is called in some literature, STRIPS planning [5]. Classical planning is lightweight in the sense that it is easier to get into the theory of planning (independently on a representation of planning, which will be discussed in this chapter) and it is easier to describe domains of problems. The concept of classical planning is similar to the theory of Automata and Grammars (deterministic finite automaton (DFA), to be precise). There are different states of the world as well as actions that change one state to another. An action in classical planning can be viewed as a state transition function $\gamma : S \times A \to S$, where S and A denote the sets of all states and actions, respectively. If an action is applicable to a state then there will be a state transition, otherwise it will be undefined. The set of goal states, $S_g \subseteq S$, is analogous to a set of accepting states. The planning problem then can be reduced to a problem of searching paths in a directed graph.

Although this approach to classical planning is simple and straightforward, it has one crucial flaw which makes it unusable in practice. Imagine a planning domain that describes a warehouse. If we have five locations, three piles of containers per location, three robots, and one hundred containers, then there are 10^{277} states [6], which is much more than the number of atoms in the observable universe. And that is just a simple small domain with a small amount of objects. For this purpose, we need to use better ways of representing classical planning, which will be called *state-transition system*. We will briefly describe two possible representations: *set-theoretic representation* and *classical representation*.

In both representations, the idea is to use more compact ways of defining a state of the world and actions that change the state. Instead of having a set of all states S, we will have a finite set of proposition symbols $P = \{p_1, \ldots, p_n\}$ in *set-theoretic representation* and a first-order language $\mathcal{L}$ in *classical representation*. Both, P and $\mathcal{L}$ describe properties and features of the world. Each state is then defined by a set of properties, which are true in that state and every other property is false. This feature is called closed world assumption. Both of these representations have the same expressive power [6].

All of the following definitions are correspondent and equivalent to the definitions in the book *Automated Planning: theory and practice (Chapter 2)* [6].

1.1 Set-Theoretic Representation

Definition 1. *Let $P = \{p_1, \ldots, p_n\}$ be a finite set of propositional symbols, which will model properties of a world. Then $\Sigma = (S, A, \gamma)$ is a* planning domain *on P. S, A, and γ denote the set of all states, set of all actions, and the* state-transition function*, respectively, such that:*

- *$S \subseteq 2^P$, each state $s \in S$ is described by propositions that hold in that state. Other propositional symbols $p \in P$, $p \notin s$ do not hold in a state s.*

- *Each action $a \in A$ is defined as a triple of sets describing the preconditions and changes to a current state. Action $a \in A$ will be denoted as $a = (pre(a), eff^-(a), eff^+(a)) \in 2^P \times 2^P \times 2^P$, where $pre(a)$ stands for preconditions that must be true in a state to apply this action, meanwhile $eff^-(a)$ and $eff^+(a)$ describe changes to the state or, in other words, negative and positive effects*[1]*. We require $eff^-(a) \cap eff^+(a) = \emptyset$ for every $a \in A$, because otherwise it does not make any sense. Having a state $s \in S$, an action $a \in A$ is* applicable *to state s, if and only if $pre(a) \subseteq s$ (it might be convenient to specify $pre^-(a)$ and $pre^+(a)$ to denote positive and negative preconditions).*

- *Having $s \in S$ and an action $a \in A$ that is* applicable *to state s,* state-transition *function γ is defined as follows: $\gamma(s, a) = (s - eff^-(a)) \cup eff^+(a)$, otherwise it is undefined.*

- *S must have a property that for every $s \in S$ and $a \in A$ that is* applicable *to s, $\gamma(s, a) \in S$. This way we do not have to know all of the states in advance. All we need is a set of actions A and some initial state $s_0 \in S$, then we can find all reachable states using Breadth-First Search (BFS), Depth-First Search (DFS), or some other searching algorithms.*

Definition 2. *A* planning problem *is a triple $\mathcal{P} = (\Sigma, s_o, g)$, such that:*

- *$\Sigma = (S, A, \gamma)$ is a* planning domain *on P.*

- *$s_0 \in S$ is an* initial state*.*

- *$g \subseteq P$ is a set of* goal proposition symbol*, which must be true in the state after the final action of a plan. $S_g = \{s \in S | g \subseteq s\}$ is a set of goal states, i.e., states that satisfy the* planning problem *$\mathcal{P}$.*

Because the set of *propositional symbol* P is finite, sets of states, actions, and state-transition function are also finite.

Definition 3. *A* plan *in a* planning domain *Σ is a finite sequence of actions $\pi = (a_1, \ldots, a_k)$, where $k \geq 0$ and $(\forall a \in \pi) a \in A$. The* plan *$\pi$ is* applicable *to a state $s_0 \in S$, if and only if a sequence of states $(s_0, \ldots, s_k)$ exist, such that: $(1 \leq i \leq k)$ $pre(a_i) \subseteq s_{i-1}$ and $s_i = \gamma(s_{i-1}, a_i) = ((s_{i-1} - eff^-(a_i)) \cup eff^+(a_i))$ is defined. Otherwise, the* plan *π is invalid.*

In other words, a plan *$\pi = (a_1, \ldots, a_k)$ is* applicable *to a state s_0, if and only if a* plan *$\pi' = (a_2, \ldots, a_k)$ is* applicable *to a state $s_1 = \gamma(s_0, a_1) \in S$.*

Having a plan *$\pi = (a_1, \ldots, a_k)$ and a state $s \in S$, we will abuse the notation to let $\gamma(s, \pi)$ denote a state $s_k \in S$, where we get after applying the sequence of actions of a plan, i.e.,*

$$\gamma(s, (a_1, \ldots, a_k)) = \begin{cases} s, & \text{if } k = 0 \text{ (there are no actions)}, \\ \gamma(\gamma(s, a_1), (a_2, \ldots, a_k)), & \text{if } k > 0, \ a_1 \text{ is applicable to } s, \\ undefined, & \text{otherwise.} \end{cases}$$

[1] In some literature these sets are called: add list, delete list.

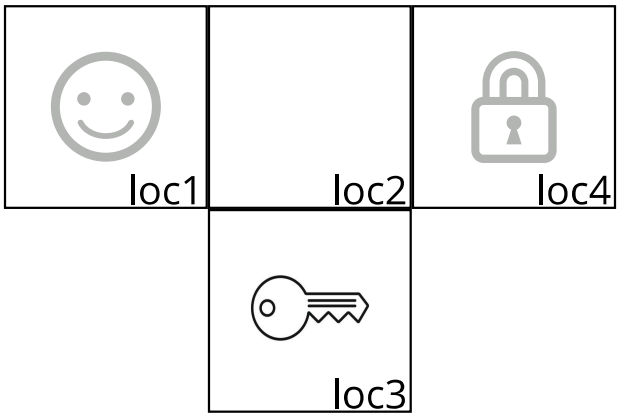

Figure 1.1 Initial state of a simple planning domain

Definition 4. *A* solution *to a* planning problem $\mathcal{P} = (\Sigma, s_o, g)$ *is a plan* $\pi = (a_1, \ldots, a_k)$ *that is* applicable *to* s_0*, which changes the initial state* s_0 *to a state that satisfies* goal propositional symbols*:* $\gamma(s_0, \pi)$ *is defined and* $g \subseteq \gamma(s_0, \pi)$ *(equivalent to* $\gamma(s_0, \pi) \in S_g$*).*

For a *planning problem* there might be many *solutions*, moreover, there might be an infinite number of *solutions* if there are states that can be visited repeatedly, (e.g. we might have two adjacent locations in a domain, that can be visited cyclically without any constraints).

It is easy to see that for each *planning problem* $\mathcal{P}$ with nonempty set of *solutions* (there exist a *plan* π such that π is a solution to a $\mathcal{P}$) there exist at least one *minimal solution* $\pi = (a_1, \ldots, a_k)$, i.e., there is no *solution* $\pi' = (a'_1, \ldots, a'_j)$ such that $j < k$. There might be many different *minimal solutions* but all of them share the same length.

Example 1. *Imagine a simple planning domain (Figure 1.1) that can represent states in an uncomplicated maze-like game. We have four game locations, a player character, a key, and a locked door (illustrated by a* loc4*). The character can move freely between locations but if he wants to enter* loc4*, then he needs a key that can be picked at the* loc3 *(but cannot be discarded later on). Now we will define a planning problem* $\mathcal{P} = (\Sigma, s_0, g)$ *for this domain* $\Sigma = (S, A, \gamma)$ *more formally:*

- $P = \{$at-loc1,at-loc2,at-loc3,at-loc4,key-picked$\}$,
- $S = \{\{$at-loc1$\}, \{$at-loc2$\}, \{$at-loc3$\}, \{$at-loc1, key-picked$\}$,
  $\{$at-loc2, key-picked$\}, \{$at-loc3, key-picked$\}, \{$at-loc4, key-picked$\}\}$,

- $A = \{\text{move12}, \text{move21}, \text{move23}, \text{move32}, \text{move24}, \text{move42}, \text{pick}\}$, where :
  $\text{move12} = (\{\text{at-loc1}\}, \{\text{at-loc1}\}, \{\text{at-loc2}\})$;
  $\text{move21} = (\{\text{at-loc2}\}, \{\text{at-loc2}\}, \{\text{at-loc1}\})$;
  $\text{move23} = (\{\text{at-loc2}\}, \{\text{at-loc2}\}, \{\text{at-loc3}\})$;
  $\text{move32} = (\{\text{at-loc3}\}, \{\text{at-loc3}\}, \{\text{at-loc2}\})$;
  $\text{move24} = (\{\text{at-loc2}, \text{key-picked}\}, \{\text{at-loc2}\}, \{\text{at-loc4}\})$;
  $\text{move42} = (\{\text{at-loc4}\}, \{\text{at-loc4}\}, \{\text{at-loc2}\})$;
  $\text{pick} = (\{\text{at-loc3}\}, \{\}, \{\text{key-picked}\})$,
- γ as defined in Definition 1,
- $s_0 = \{\text{at-loc1}\}$ (Figure 1.1),
- $g = \{\text{at-loc4}, \text{key-picked}\}$.

Example 1 shows us a complete description of a *planning domain* Σ and a *planning problem* $\mathcal{P}$. There is one and only *minimal solution* which is $\pi = (\text{move12}, \text{move23}, \text{pick}, \text{move32}, \text{move24})$. On the other hand, there is an infinite number of *solutions*. We might go to the loc3 and apply the action pick as many times as we desire, thus generating an infinite amount of *solutions*.

With knowledge of the particular domain, we can adjust the domain without losing any domain's specifics. For example, we might alter the set of the *goal proposition symbols* g to be $g = \{\text{at-loc4}\}$ because we know that it is impossible to enter loc4 without having a key picked up at the loc3.

The set of states S contains all reachable states from the initial state s_0, other unreachable states might be added to the S but they are not needed. For example, the state {at-loc1,at-loc2} is contradictory because we cannot be at two locations simultaneously. Likewise states {}, {key-picked}, {at-loc4} are also meaningless.

The concept of the key in this domain is depicted as *a propositional symbol* key-picked. This simplification does not take into account the location of a key because it is not needed in this domain. Once the key is picked, it will be implicitly located at the same location as the character. Hence, the action pick is *applicable* more than once in a *plan*.

Example 2. *Let's consider an extension of this domain by adding two new locations, and another key (Figure 1.2). Only this time we want to model a new constraint that allows at most one key to be held at a time. There are numerous ways of modeling this planning domain. For instance, we might introduce new* propositional symbols*:* key1-picked, key2-picked, door1-opened, door2-opened. *If we wanted to model the locations of keys, we would need to include many new* propositional symbols *of the form* key1-at-loc1, key1-at-loc2*, and so on. Also, we would need new* actions *that would consider the locations of a key and the character. New actions may look like this:* pick-key1-at1 = ({at-loc1,key1-at-loc1},{key2-picked},{key1-picked}), move12-with-key1 = ({at-loc1,key1-at-loc1},{at-loc1,key1-at-loc1},{at-loc2,key1-at-loc2}) *and so on. This enumeration of possible substates might result in an enormous set of* propositional symbols *and other related sets* [6] (2.2.4 Properties of the Set-Theoretic Representation)*. An elaborate solution that fixes this problem is discussed in the following chapter.*

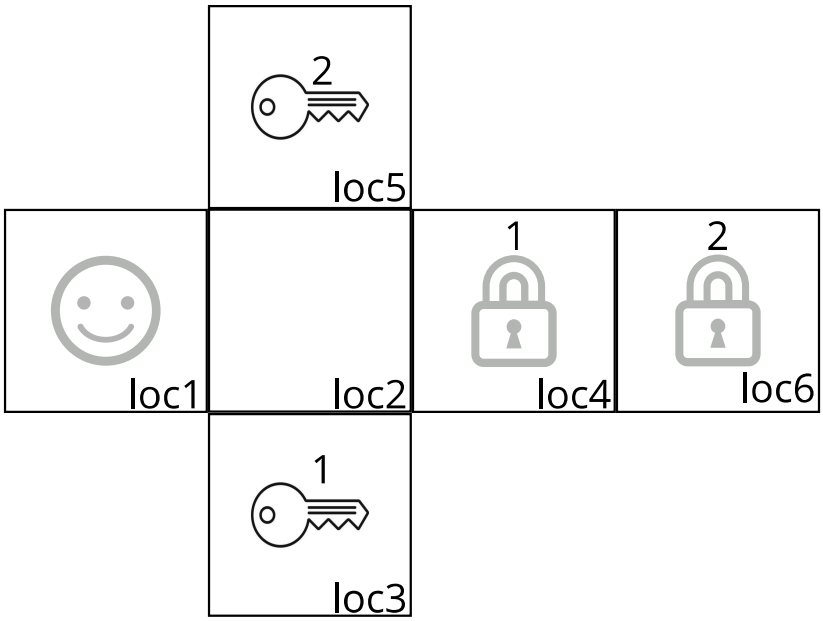

Figure 1.2 Extended planning domain of a maze-like game

1.2 Classical Representation

Classical representation of planning improves *set-theoretic representation* by introducing a first-order language $\mathcal{L}$ that consists of a finite number of constant symbols (similar to a set of *propositional symbols* defined in Definition 1), predicate symbols, and other symbols for the representation of *planning operators*. We do not allow any functional symbols in $\mathcal{L}$ because we are focused to model finite models where functions are not needed. A state in *classical representation* is defined as a set of grounded logical atoms of $\mathcal{L}$.

Example 3. *Before we start with definitions, we will look at Figure 1.1 from the perspective of* classical planning*. We want to model the player, key, locations, and locked door. Thus, the set of constant symbols is* {player,loc1,loc2,loc3,loc4 ,key1,nothing,door}*. The constant symbol* nothing *represents that the player's inventory is empty. With this knowledge, we can represent the initial state of Figure 1.1 to be* $s_0 = \{\text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{adjacent}(\text{loc2}, \text{loc3}), \text{adjacent}(\text{loc3}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc4}), \text{adjacent}(\text{loc4}, \text{loc2}), \text{at}(\text{player}, \text{loc1}), \text{holds}(\text{player}, \text{nothing}), \text{locked}(\text{door})\}$*. Some logical atoms will remain the same throughout all possible states, e.g.* adjacent(loc1,loc2)*, meanwhile other atoms like* at(player,loc1) *or* holds(player,nothing) *will change after the application of proper actions.*

Now we will define *planning operators* in *classical planning*. A grounded instance of an operator is an action that can be part of a *plan*.

Definition 5. *A* planning operator *is a triple* $o = (name(o), precond(o), eff(o))$*, where:*

- $name(o)$ *is of a form* $n(x_1, \ldots, x_k)$*, where* n *is a* operator symbol *and must be unique in* $\mathcal{L}$ *and* $x_1, \ldots, x_k$ *represent variable symbols that appear in an operator,*
- $precond(o)$ *and* $eff(o)$ *generalize preconditions and effects of* set-theoretic representation*. These sets contain positive and negative logical literals.*

Let O *denote a set of all* planning operators *as specified above.*

In some literature, the usage of $name(o)$ is not specified, rather the multiset of grounded operators (actions) is assumed. The essence of $name(o)$ is in the planning phase. We might have multiple grounded instances of operators that might use the equivalent constant symbols. To help differentiate a *planning operator* from an instance of a *planning operator*, the $name(o)$ might come in handy. $name(o)$ refers unambiguously to one specific *planning operator*. We will omit the usage $name(o)$ because the meaning will be obvious from the context.

Example 4. *In this example, we will look at some possible* planning operators *that might be used in Figure 1.2 (we will use new predicate symbols that were not specified yet):*

- move(player,location1,location2)
  precond: adjacent(location1,location2), at(player,location1), ¬locked(location2)
  effects: ¬at(player,location1), at(player, location2)
- pickup-key(player,key,location)
  precond: at(player,location), at(key,location), holds(player,nothing)
  effects: ¬holds(player,nothing), ¬at(key,location), holds(player,key)
- drop-key(player,key,location)
  precond: at(player,location), holds(player,key)
  effects: ¬holds(player,key), at(key,location), holds(player,nothing)
- open-door(player,location1,location2, key)
  precond: adjacent(location1,location2), locked(location2), at(player,location1), key-opens-door(key,location2), holds(player,key)
  effects: ¬locked(location2)

In the Example 4 predicate symbols adjacent(l1,l2) and key-opens-door(key,location) are describing static properties of a domain that will never change and thus, will never appear in effects of *planning operators*. On the other hand, predicate symbols like at(player/key,location), locked(location), or holds(player, key) will change and therefore will occur in effects of *planning operators*.

Definition 6. *Let us be given a state $s \in S$ and an action (ground instance of a* planning operator*) a. If $precond^+(a) \subseteq s$ and $precond^-(a) \cap s = \emptyset$, where $precond^+(a)$ denotes a set of all positive atoms in $precond(a)$ and $precond^-(a)$ denotes a set of all atoms whose negation is in $precond(a)$, then a is* applicable *to s and $\gamma(s,a) = (s - eff^-(a)) \cup eff^+(a)$. $eff^-(a)$ and $eff^+(a)$ are analogous to $precond^-(a)$ and $precond^+(a)$, respectively.*

Definition 7. *Having a first-order language $\mathcal{L}$ with a finite amount of constant and predicate symbols and no functional symbols, the* classical planning domain *in $\mathcal{L}$ is a state-transition system $\Sigma = (S, A, \gamma)$ such that:*

- $S \subseteq 2^{\{all\ ground\ atoms\ of\ \mathcal{L}\}}$,
- $A = \{all\ ground\ instances\ of\ operators\ in\ O\}$,
- γ *is analogous Definition 1,*
- S *is closed under γ in the same manner as in Definition 1.*

Definition 8. *A* classical planning problem *is corresponding to* set-theoretic planning problem*, it is defined as $\mathcal{P} = (\Sigma, s_0, g)$, where:*

- $\Sigma = (S, A, \gamma)$ *is a* classical planning domain*,*
- $s_0 \in S$ *is an initial state*
- g *is a set of literals.*

2 Hierarchical Planning

Now that we have understood classical planning, we can move on to hierarchical planning or Hierarchical Task Network (HTN), to be precise. In HTN the goal alone is not to find an *applicable plan* that will meet all of the criteria of the *planning problem*. HTN focuses on accomplishing some set of tasks by decomposing them into sub-tasks with constraints like ordering and state conditions. A task is accomplished immediately upon all of his sub-tasks are accomplished. HTN is an extension of classical planning which gives us clues on how to find the proper *plan*.

As in classical planning, there are states of the world represented by the set of *proposition symbols* or *logical atoms* that are true in a state. We can *apply actions* to states if the preconditions of *actions* are satisfied (held in a state). These *actions* will be called *primitive tasks* in HTN. These tasks are executable right away and they change the current state to another one by applying negative and positive effects. In addition to *primitive tasks* there are *non-primitive tasks*. We will call them *compound tasks*. These tasks cannot be executed and *applied to a state*, rather they have to be decomposed into a set of sub-tasks (both *primitive* and *compound*) using decomposition *methods*. Having an *initial task network*, that is a set of tasks and a set of constraints, the goal is to decompose all tasks into *primitive tasks* and find an ordering of these tasks so that the constraints are not violated. After doing so, it is important to check that the resulting *plan* (a sequence of *primitive tasks*) is a valid *plan* and can be *applied* to an *initial state*.

All of the following definitions are correspondent and equivalent to the definitions in the book *Automated Planning: theory and practice (Chapter 11)* [6] and other publications [3] [4] [7] [8].

2.1 HTN Planning

Definition 9. *A* task network *is a pair* $\omega = (U, C)$*, where* U *is a set of* tasks *(primitive and compound) and* C *is a set of constraints.*

For a solution *(*plan*) to be valid we need to satisfy all of the constraints listed in* C*. We identify four types of constraints:*

- $t_i \prec t_j$ *where* $t_i, t_j \in U$*: an* ordering-constraint *meaning that in every valid* solution *the last* primitive task *(*action*) in the* solution *to which* t_i *decomposes must precede the first* primitive task *(*action*) in the* solution *to which* t_j *decomposes,*
- *before*(l, U')*:* before-constraint *symbolizing that in every valid* solution *the literal* l *must hold in a state right before the* application *of a first* action *to which* $U' \subseteq U$ *decomposes,*
- *after*(U', l)*:* after-constraint *is similar to* before-constraint *with the difference that the literal* l *must hold in a state right after the last* action *to which set* U' *decomposes,*

- *between*(U', l, U'')*:* between-constraint *is saying that the literal* l *must hold in all states starting after the last* action *of* U' *and lasting until the state right before the* application *of a first* action *of* U''*.*

We will say that the task network *is* primitive *if and only if, all of the tasks are* primitive*. A* task network *having at least one* compound task *will be called* non-primitive*.*

The most used *state-constraint* is *a before-constraint* which can be modeled in various HTN planners. The usage and application of this constraint is obvious. Before the beginning of a task, we want to hold some preconditions that are significant for that particular task. For example, a user needs to input all of the information before he can continue with the paying; a player needs to complete all of the quests before he can move on to the next stage, or a warehouse robot needs to have permission set before he can execute some real-life tasks. Analogous *after-constraint* has different use cases. We might use this constraint on the last task in *the task network* to signalize our intentions which are not modeled with the tasks themselves. For example, we want to deallocate resources of the process after the completion, or we want to give a player new abilities after the completion of many tasks that might interleave. This constraint can be used as a shortcut in difficult models where the person responsible for the modeling cannot enumerate and decide the ordering of tasks perfectly. *The between-constraint* represents an interval of states which hold some property. For example, we want a player to have a specific item in his inventory after the completion of one task and lasting until the other tasks, or an autonomous robot must hold a crate in all states meanwhile he moves from one location to another (possibly doing other unrelated things).

In accordance with the Definition 5, the more formal definition [2] [3] [6] of the *task network* would contain some function $\alpha : U \to N$, which would map instances of *primitive* and *compound tasks* to the set of names of all *tasks*. This function would allow us to have multiple instances of the same *task* in the set of tasks U, each with a unique identifier. We will omit a function α because its context will always be clear.

Definition 10. *A* method *is a triple* $m = (compound(m), subtasks(m), constraints(m))$*, where* $compound(m)$ *is a* compound task *and* $(subtasks(m), constraints(m))$ *is a* task network*.* $constraints(m)$ *operate only with the tasks in* $subtasks(m)$*. Having a* task network $\omega = (U, C)$*,* compound task $c \in U$*, and an instance of* method m *from* M *such that* $compound(m) = c$*. Then* m *decomposes* c *into* $subtasks(m)$*, creating a new* task network $\omega' = (U', C')$ *of a form:*

$$\delta(\omega, c, m) = ((U - \{c\}) \cup subtasks(m), C' \cup constraints(m)),$$

where C' *is a modification of* C*:*

- *replace each* ordering-constraint *containing* c *with new ones containing* $subtasks(m)$*, i.e.,* $\forall c' \in subtasks(m) : c \prec x$ *is replaced with* $c' \prec x$ *and* $x \prec c$ *is replaced with* $x \prec c'$*,*

- *replace each* before-, after-, between-constraint *containing* c *with new ones containing* $subtasks(m)$. *For example, we would replace* before-constraint $before(l, V)$ *with* $before(l, (V - \{c\}) \cup subtasks(m))$.

It is highly convenient to write planning *methods* as rewriting rules [7]:

$$T \to T_1, \ldots, T_k \quad [C].$$

This rewriting rule symbolizing a *method* is saying as follows: A *method* which decomposes the *compound task* T into sub-tasks $T_1, \ldots, T_k$ under the constraints C.

Definition 10 works well for *planning domains* without *empty methods* (a *method* that decomposes compound task into the empty set of subtasks). Using *an empty method* creates undefined behavior because the decomposed compound task is removed from all constraints and thus it is unclear where the specified constraint should be checked. For this reason, we will later redefine *method* decomposition.

Definition 11. *A* (HTN) planning domain *is a quadruple* $\mathcal{D} = (\mathcal{L} \ / \ P, O, C, M)$, *where* $\mathcal{L} \ / \ P$ *and* O *are first-order language (*set of propositional symbols*) and a set of* operators*, respectively, as described in the section Classical Representation,* C *is a set of* compound tasks *and* M *represents a set of all* methods.
Planning domain *will be referred to as* totally-ordered *if all of the* methods *in* M *and the* initial task network *are* totally-ordered *(there is a total strict ordering of sub-tasks).* Planning domains *that are not* totally-ordered *will be referred to as* partially-ordered.

A first-order language $\mathcal{L}$ can be easily interchanged for a set of *propositional symbols* P due to the equivalent expressive power of *set-theoretic representation* and *classical representation* [6].

Definition 12. *A* (hierarchical) planning problem *is a triple* $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *that consists of* initial state *of the world* s_0, initial task network ω *and a* planning domain $\mathcal{D}$.

Definition 13. *A* solution *to the* planning problem $\mathcal{P}$ *is a* plan $\pi = (a_1, \ldots, a_k)$. *This sequence of* actions *is assembled from a* primitive task network $\omega' = (U', C')$, $U' = \{a_i \mid 1 \leq i \leq k\}$, *after the application of a sequence of* methods *from* M *to the* initial task network ω. *A* solution π *needs to be valid, i.e.,* $\gamma(s_0, \pi)$ *is not undefined and also needs to satisfy all of the constraints in* C'. *That is:*

- $a_i \prec a_j \implies i < j$,
- $before(l, V) \implies l \in s_{min\{i \mid a_i \in V\} - 1}$ *(if the literal* l *is negative then it must not be in a state, same applies for other constraints),*
- $after(V, l) \implies l \in s_{max\{i \mid a_i \in V\}}$,
- $between(V', l, V'') \implies (\forall j,\ max\{i \mid a_i \in V'\} \leq j < min\{i \mid a_i \in V''\} : l \in s_j)$.

By l *we mean a ground instance of a literal* l *(in the case of* set-theoretic *representation it would be* $p \in P$*).*

2.2 HTN and Context-free Grammars

With the knowledge acquired in the previous sections, we can try to map planning definitions and structures to the definitions and structures of the theory of Automata and Grammars. By doing so, we would be able to utilize already discovered knowledge and algorithms. Furthermore, it will allow us to prove different language classifications [3].

The simplest example is to map *classical planning problem* to the equivalent DFA, and vice versa. The mapped DFA generates all of the *solutions*, i.e., *plans* that are valid, *applicable* to the *initial state* s_0 and satisfy the goal g.

Theorem 1. *For every* classical planning problem $\mathcal{P} = (S, A, \gamma, s_0, g)$*, where* (S, A, γ) *is a* planning domain*, there exists a DFA [1]* $D = (Q, \Sigma, \delta, q_0, F)$ *such that* D *generates all the* solutions *for* $\mathcal{P}$*.*

Proof. The proof is straightforward. Having $\mathcal{P} = (S, A, \gamma, s_0, g)$, we can construct DFA $D = (S, A, \delta, s_0, S_g)$ where $S_g = \{s \in S | g \subseteq s\}$ and $\delta(s, a) = \gamma(s, a) = (s - \text{eff}^-(a)) \cup \text{eff}^+(a)$. We can easily see that every generated word (*plan*) $\pi \in L(A)$ is also a valid *solution* to the *planning problem*. Every *solution* is *applicable* to the $q_0 = s_0$ because *transition function* of D imitates *planning state-transition function* of $\mathcal{P}$. Every $\pi \in L(A)$ is a *solution* because the set of accept states is S_g. □

Theorem 2. *For every DFA* $D = (Q, \Sigma, \delta, q_0, F)$ *there exists a correspondent* classical planning problem $\mathcal{P} = (S, A, \gamma, s_0, g)$ *where the set of* solutions *of* $\mathcal{P}$ *is equal to* $L(D)$*.*

Proof. Having DFA $D = (Q, \Sigma, \delta, q_0, F)$, we will construct *planning problem* $\mathcal{P}$ with individual components defined as follows. Without loss of generality, we will rename states of Q. Each state $q \in Q$ will have a unique index i: $0 \leq i < |Q|$. q_0 is the *initial state* of D. Now we define $\mathcal{P} = (S, A, \gamma, s_0, g)$ with:

$$
\begin{gathered}
S = \{\{i\} | q_i \in Q - F\} \cup \{\{i, goal\} | q_i \in F\}, \\
A = \{\sigma = (\{i\}, \{i, goal\}, \{j\}) | \delta(q_i, \sigma) = q_j \text{ and } q_j \notin F\} \cup \\
\{\sigma = (\{i\}, \{i\}, \{j, goal\}) | \delta(q_i, \sigma) = q_j \text{ and } q_j \in F\}, \\
\gamma(s, a) = (s - \text{eff}^-(a)) \cup \text{eff}^+(a), \\
s_0 = \begin{cases} \{0\}, & \text{if } q_0 \notin F, \\ \{0, \text{goal}\}, & \text{if } q_0 \in F, \end{cases} \\
g = \{goal\}.
\end{gathered}
$$

Every generated word $\sigma_1 \dots \sigma_n \in L(D)$ corresponds to a single *solution* $\pi = (a_1, \dots, a_n)$. Each state of $\mathcal{P}$ is represented by an index (*propositional symbol*) i which specifies the state $q_i \in Q$. If a state is an accepting state, then the planning state s has *propositional symbol goal*. This is forced by the positive and negative effects of *actions*. □

Corollary. Classical planning is classified into the class of regular languages.

Now we will prove that *totally-ordered* HTN planning can be represented as a context-free (CF) grammar and vice versa. These claims are extremely useful and will be used in the following parts of this thesis. Similar proofs are presented in [3].

Theorem 3. *For every CF grammar* $G = (V_G, T_G, P_G, S_G)$ *[1] there exists a correspondent* totally ordered planning problem $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *with* $\mathcal{D} = (P, O, C, M)$ *such that the set of* solutions *of* $\mathcal{P}$ *is equal to* $L(G)$.

Proof. Constructing *planning problem* $\mathcal{P}$ is uncomplicated and intuitive because the structures are similar. All we need is to handle *operators* $o \in O$ so that the generated *plans* are *applicable* to the *initial state* s_0. Having $G = (V_G, T_G, P_G, S_G)$ we construct $\mathcal{P} = (\{\}, (\{S_G\}, \{\}), \mathcal{D})$ with $\mathcal{D} = (\{\}, T_G, V_G, M)$. *Methods* are defined as follows:

$$M = \{T \to T_1, \ldots, T_k \; [\; \{T_i \prec T_j \mid 1 \leq i < j \leq k\} \;] \mid (T, T_1 \ldots T_k) \in P_G\}.$$

Using this construction, the *planning problem* $\mathcal{P}$ can now generate the set of *plans* that equals $L(G)$. One problem is that the generated *plans* might not be valid *solutions*. This can be handled conveniently if we use empty sets for the preconditions of all *operators*. Each $o \in O$ will be defined as $o = (\{\}, \{\}, \{\})$. The result of this modification is that every possible *plan* is now a valid *solution* if it can be decomposed from the *initial task network*. □

Theorem 4. *For every* totally ordered planning problem $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *with* $\mathcal{D} = (P, O, C, M)$ *without* before-, after-, between-constraints *there exists a CF language* L *that is equal to the set of* solutions *of* $\mathcal{P}$.

Proof. Let $\mathcal{P} = (s_0, \omega = (U, C), \mathcal{D} = (P, O, C, M))$ be a *totally ordered planning problem* in which constraints of each *method* consist only of *ordering-constraints*. We will construct CF grammar $G = (V_G, T_G, P_G, S_G)$ where $V_G = C, T_G = O$ and $S_G \in U$. We assume that the *initial task network* ω consists of one task, $|U| = 1$. If there are more tasks (that are totally ordered by constraints C), we will add one extra production rule to the P_G that rewrites the new starting symbol S'_G to the given state of $\omega = (U, C)$. The set of production rules is defined as follows:

$$P_G = \{(T, T_1 \ldots T_k) \mid (T, \{T_1, \ldots, T_k\}, \{T_i \prec T_j \mid 1 \leq i < j \leq k\}) \in M\}.$$

Grammar G generates all *plans* that can be generated with $\mathcal{P}$. Some of these *plans* might be valid *solutions* and some of them might not be *applicable* to the *initial state* s_0. In other words, $L(G)$ is a superset of all possible *solutions* of $\mathcal{P}$. To solve this issue it is sufficient to use the fact that the intersection of a context-free language and a regular language is resulting into context-free language [1]. We can use Theorem 1 to build a DFA D that will generate all *plans* that are *applicable* to s_0. The empty set of goal symbols $g = \emptyset$ will ensure that the language of D equals to all possible *plans* that are allowed by the planning *operators* O. Having everything prepared, we can construct a CF language $L = L(G) \cap L(D)$ which is equal to the set of *solutions* of $\mathcal{P}$. □

Corollary. Totally-ordered HTN planning without state-constraints is classified into the class of context-free languages.

2.3 HTN Plan Verification

Hierarchical plan verification can be seen as a reversal process to the decomposition of a *initial task network* into a *plan* that is *applicable* to the *initial state* s_0 and satisfies all constraints. Given a *hierarchical planning domain* $\mathcal{D} = (P, O, C, M)$, *initial task network* ω, an *initial state* s_0, and a *plan* π, we ask if it is possible to decompose the ω into π using $\mathcal{D}$. Depending on a hierarchical model or semantics specification, there might be some additional information and structures to the input of a plan verification problem. For example, we might analyse totally-ordered plan verification with *method* preconditions [4].

2.4 HTN Semantics with Goal Tasks

Up to now, we have only discussed HTN definitions based on the book *Automated Planning: theory and practice* [6]. This book does not consider *goal tasks* [2] which are very similar to the *goals* in classical planning (Definition 2). *Goal tasks* are attached to the *compound tasks* and define the set of *propositional symbols* (or logical atoms) that need to hold in a state after the execution of all *primitive tasks* to which *the compound task* decomposes. In the case of *primitive task*, the set of positive effects is logically equivalent to the *goal task* of that task.

Many formalisms for describing hierarchical planning exist, and each of them comes with different approaches to define and propose *hierarchical planning problems*. Each formalism allows diverse features that might come in handy during modeling or planning of the *planning problems*. For example, HDDL [9], allows to specify *goal state* which is equivalent to the set of *goal predicate symbols* defined in Definition 2. For a *plan* to be valid in HDDL, a sequence of *task decompositions* transforming *the initial task network* into *the primitive task network* must exist. Also, *the primitive task network* must satisfy all constraints, a linear ordering of *primitive tasks (actions)* must exist, and the state after the execution of the *plan* must hold *propositional symbols* defined in *goal state*. This is a great feature but the definition of *a goal state* is allowed only for the whole *planning problem* and not for the individual tasks. On the other hand, alternative formalism [2] grant the feature of *goal tasks* which is noticeably more flexible than HDDL's formalism.

Now we will present a way of using *goal tasks* in modeling and planning. The idea is to omit the negative effects of *actions*. This way, all *actions* will only append *predicate symbols* to the state (which is defined as an element of the set 2^P). In such a manner, the planning might remind us of monotone functions. The sequence of intermediate states $(s_0, \ldots, s_k)$, created by the *plan* $\pi = (a_1, \ldots, a_k)$, has a non-decreasing size of *propositional symbols* in a state. Thus, we can call this subclass of *planning problems – monotone hierarchical planning*.

With the knowledge from the previous paragraph, let's combine the concepts of *goal tasks*, *monotone hierarchical planning*, and totally-ordered *planning domains*. Assume a *compound task* T is decomposed into the subtasks $T_1, \ldots, T_k$ (with the ordering $T_1 \prec \cdots \prec T_k$) with given *goal tasks*. Because of the totally-ordered domain and the *monotone hierarchical planning*, we can use the technique divide and conquer and try to find the decompositions of subtasks independently.

Furthermore, the subtask T_i might take into consideration *propositional symbols* contained in *goal tasks* of subtasks $T_1, \ldots, T_{i-1}$. These symbols can be viewed as granted because of the fact that there are no negative effects and all methods are totally-ordered.

3 HTN Semantics

So far we have only discussed the definitions of classical and hierarchical planning. In this chapter, we will consider various semantics for hierarchical planning. A large part of this chapter will compare and describe semantics concerning *empty methods* which are not defined properly within Definition 10. The main issue regarding *empty methods* is caused by the unclear and ambiguous modification of *task network's* constraints. It is uncertain how and when the modified constraints need to be checked. In the first part of this chapter, we will analyze existing approaches and propose new ways of handling *empty methods* concerning the HTN plan verification process. In the second part, we will introduce a new idea for modeling hierarchical planning problems without the negative effects of *primitive tasks*.

3.1 Empty Methods

The following section will describe different solutions of how to handle *empty methods*.

Example 5. *To understand the importance of* empty methods *we will reuse Figure 1.2 as an example.*

Suppose that the game's initial state *is* $s_0 = \{at\text{-}loc1, key2\text{-}at\text{-}loc5, key1\text{-}at\text{-}loc3, door1\text{-}locked, door2\text{-}locked, key1\text{-}picked\}$. *We will assume that multiple keys can be held at the same time, and, for some reason, the player has already picked up the key*1 *(for example he picked it up in the previous level).* The initial task network *is* $(\{Get\text{-}to\text{-}loc6\}, \{\})$. *An arrow* $\rightarrow$ *symbolizes* a decomposition.

$$(\{Go\text{-}to\text{-}loc6\}, \{\}) \rightarrow (\{Get\text{-}key1, Get\text{-}key2, Go\text{-}to\text{-}loc4, Go\text{-}to\text{-}loc6\}, \{Get\text{-}key1 \prec Go\text{-}to\text{-}loc4, Get\text{-}key2 \prec Go\text{-}to\text{-}loc6, before(key1\text{-}picked, Get\text{-}key2\})$$

At this point it is very convenient to use an empty method *that decomposes* $Get\text{-}key1$ *into nothing because we have already picked up the key.*

Example 6. *Suppose we have a simple* task network $\omega = (\{t1, t2\}, \{t1 \prec t2, before(p, t2)\})$. *In natural language, we would say that task (*primitive *or* compound*)* $t1$ *must precede task* $t2$*, and at the same time, before executing the first* primitive task *to which task* $t2$ *decomposes (in case of a* primitive task*, we mean the state before the* application *of the task* $t2$*) the* propositional symbol p *must hold in a state. Having an* empty method m *such that:*

$$t2 \rightarrow \varepsilon \quad [C], \text{ where } C = \emptyset \text{ and } \varepsilon \text{ denotes no subtasks},$$

we can decompose $t2$ *in* task network ω*. After the application of the* empty method m*, we are not able to tell with certainty the semantics of the result* task network*. By* applying m*, we erase the task* $t2$ *from the set of tasks in* ω *and we modify all of the constraints in* C*. The unclear result might look like this:*

$\omega' = (\{t1\}, \{t1 \prec ?, before(p, ?)\})$ *with* ? *symbolizing undefined behaviour. Thus, the following section will study different ways of representing HTN planning with the goal of resolving undefined behavior of* empty methods*. Usually, we will need some additional information about* task network's *decomposition process which will deal with unclear situations. Sometimes we will need to modify existing definitions from the previous chapters.*

Another possible use case of *empty methods* concerns situations when the decomposed task is contained in *the method's* constraints set. Our definitions of decomposition do not allow this behavior, nonetheless, it might be beneficial for the architect of *a planning domain*. Suppose *a task network* containing *a compound task Pick-up-item-X*. With the extended *method decomposition* definition, we would be able to decompose this task into nothing while not saying explicitly where the state check should be made:

$$Pick\text{-}up\text{-}item\text{-}X \rightarrow \varepsilon\ [\{before(X\text{-}in\text{-}inventory, Pick\text{-}up\text{-}item\text{-}X)\}]$$

This extended definition also contains undefined behavior because it is unclear where the *state-constraint* needs to be checked. Transformations of HTN models with the extended decomposition definition can be found here [8].

3.1.1 No Empty Methods Model

The most effortless way to handle *empty methods* is to forbid them (every totalitarian dictator would approve). This way, each *method* $m \in M$ must have nonempty set of $subtasks(m)$, i.e.:

$$T \rightarrow T_1, \ldots, T_k \quad [C],\ where\ k \geq 1.$$

The main benefit of this semantics is regarding *the method* decomposition in Definition 10. By forbidding the usage of *empty methods* we are not able to create fuzzy situations with unclear constraints. This way we do not need to adjust any definition and everything works as it should.

As it happens in life, forbidding something might help in some cases but, on the other hand, it closes doors for the flexibility and advantages of *empty methods*. Now, it is not possible to fulfill the task by doing nothing which might be very convenient in states that already hold all desired *propositional symbols*. In such states, the decomposition of certain *compound tasks* is meaningless and creates *plans* that are longer and more redundant than *plans* which would allow *empty methods*.

Succeeding models will allow different kinds of *empty methods*, usually by handling some workaround.

3.1.2 No-op Based Model

Another possible way to resolve the issue with *empty methods* is the No-op model [7] [8]. The no-op model introduces a new symbol *no-op*() which is not

presented in the *hierarchical planning domain*. Formally, the *no-op*() symbol is just another *primitive task* with no preconditions and effects, i.e. *no-op*() = ({}, {}, {}). The semantics behind this symbol is hidden right in the name, "no operation", meaning that this action has no impact on the world that we model. It serves as a regular symbol during the planning phase concerning all of the *ordering-constraints* and *before-, after-, between-constraint*. After the checking of all available constraints, the *no-op*() symbol is deleted from the final *plan* as it was not part of the former *planning problem*.

Pros of the no-op model are easily seen. We have solved the issue of *empty methods* by introducing a new symbol *no-op*() and at the same time we did not need to allow regular *empty methods* which decompose a task into an empty set. The no-op model extends the previous model in Section 3.1.1.

Cons of this model lies in the HTN plan verification, reverse process to the *hierarchical planning*. The main problem is due to the fact that the *no-op*() symbols are not part of the input to the plan verification because they are removed after the check of all constraints in a *primitive task network* (consists only of *primitive tasks*). For this reason, the no-op model is inefficient for plan verification because it would be needed to guess the locations and number of no-op() symbols in the input *plan*, e.g.:

$$
\begin{gathered}
input : \pi = (task1, task2, task3) \\
guess1 : (task1, task2, task3) \\
guess2 : (task1, \textit{no-op}(), task2, task3) \\
guess3 : (\textit{no-op}(), task1, task2, task3, \textit{no-op}()) \\
guess4 : (task1, \textit{no-op}(), \textit{no-op}(), \textit{no-op}(), task2, task3)
\end{gathered}
$$

The main problem with the no-op model is the inability to verify *plan* efficiently. In the no-op model, we erase *no-op*() symbols after the checking of all constraints. It is possible to avoid this problem by forcing the planner of *planning problem* to include the *no-op*() symbol into his *planning domain* and treat it as an *empty method* (in some sense). With this practice, the planner will not need to erase *no-op*() symbols in the resulting *plan* which will make HTN plan verification much more efficient. The second version of the no-op model also extends Section 3.1.1 (No Empty Methods Model) as it does not allow real *empty methods* in *planning problems*.

Even though we have solved the *empty methods* problem and HTN verification problem, we still have not used *empty methods*, but rather some illusion of them. In the coming models, we will try to integrate actual *empty methods*, however, more planning information will be required.

3.1.3 Constraint Graph Model

A similar yet not equivalent ideas can be viewed in the book *Automated Planning: theory and practice (Chapter 11, STN Planning)* [6].

The constraint graph model will be the first in the series of models that will allow actual usage of *empty methods*. For this purpose, we will redefine *task network* and *method decomposition*. *Task network* will be represented with a directed acyclic graph (DAG) displaying the evolution of the *initial task network* into the *primitive task network*. *Method decomposition* will append new directed edges to the *task network* (graph). Each directed edge (u, v) determines the *compound task* u and task (*primitive or compound*) v to which the *compound task* u decomposes. Vertices will hold additional information about constraints. Now, let's put it all more formally.

Task network is a DAG $G = (V, E)$ in which the set of vertices V portray *primitive and compound tasks* and arcs depict *task decomposition* done by *methods*. As we know, only *compound tasks* can be decomposed, which is why edges can emerge only from vertices with *compound tasks*. In this model, only the graph's leaves can be decomposed (which are *compound tasks*), otherwise, it would be possible to decompose a task multiple times, which is unwanted. Having a (nonempty) *method* $m = (compound(m), subtasks(m), constr(m))$ and a *task network* G with a leaf symbolizing a *compound task* $compound(m)$, we can apply the *method* m to *task network* G. As a result, we get $G' = (V \cup subtasks(m),\ E \cup \{(compound(m),\ sub) \mid sub \in subtasks(m)\})$. The definition of *task decomposition* has one special case: unsurprisingly, *an empty method*. In this model, we do not want to follow practices of no-op models by forcing the planner to use a special symbol like *no-op*(). The desirable outcome is to support real *empty methods*, i.e. $T \to \varepsilon\ [C],\ C = \emptyset$, meanwhile allowing the planner to check all types of constraints completely. To fulfill this requirement an *empty method* will also create a new special vertex denoted by $\square$. This vertex serves for the sake of constraints in the *primitive task network*. E.g. having a *compound task comp* and *an empty method* $comp \to \varepsilon\ [C]\ C = \emptyset$, applying the *method* to the *task comp* would create a new vertex $\square$ and a new directed edge $(comp, \square)$. The difference between this approach and *no-op*() model approach is that here the empty vertex does not have any impact on actual *plan* (sequence of *primitive tasks*). Thus, the planner does not need to integrate or use any new symbols in the solving phase of a *planning problem*.

Now, we should talk about the constraints and their semantics in this model. Constraints, as we know them from the previous models, have their meaning in the planning phase. Instead of having one single set with all constraints in the *task network*, each vertex in the graph will have its own set of constraints. Every *task decomposition* provided by a *method* creates new vertices in the graph with **an empty set of constraints**. If a *method*, $T \to T_1, \ldots, T_k\ [C]$ has a non-empty set of constraints then (when the application of a *method* takes place) they are passed to **the set of constraints of a vertex which is being decomposed** (in our example it is the vertex with *compound task* T). Before the decomposition, the set of constraints of T was empty. For this purpose, each vertex in the graph is a pair of vertex and the set of constraints linked to that task. *Primitive tasks'* set of constraints is always **empty** due to the inability to decompose.

The constraints can be checked only when the *task network* is *primitive* and *a plan* $\pi = (a_1, \ldots, a_k)$ is assembled. At this point, we can start with the bottom-up analysis of the graph to check all of the constraints in all of the vertices. Starting

from the leaves which have no constraints in their vertices, the information (task position) about the *primitive tasks* is propagated to the parent vertices which might have some constraints. If a *compound task* has constraints then the information about *primitive tasks* alongside total-ordering created by the *plan* π (not be confused with total-, partial-order *planning domains*) can easily serve for an efficient constraint checks. Now let's put it more formally and clearly.

A plan $\pi = (a_1, \ldots, a_k)$ (with the parallel states $(s_0, \ldots, s_k)$) is created from *the primitive task network*, i.e. all leaves of the graph are either empty vertices $\square$ or vertices representing *primitive tasks*. *The plan* π is created only from *the primitive tasks* but before that, we need to find a total ordering of **all** leaves, including empty vertices $\square$. For that, each leaf is assigned a natural number ranging from 1 to the number of leaves. Having a leaf vertex v, its ordering can be viewed with the function $order(v)$. Along the actual *plan* π we also need *a preplan* ϕ, that is *a plan* that also contains empty vertices in the proper places. ϕ is created with the *order* function. Having $\phi = (x_1, \ldots, x_l)$, we can create a parallel series of states $(s_0, \ldots, s_l)$ such that $\gamma(s_{i-1}, x_i) = s_i$ if x_i is *an action*, and $s_{i-1} = s_i$ if x_i is an empty vertex. This *preplan* is then used to check constraints.

Let $m = T \rightarrow T_1, \ldots, T_k\ [C]$ be a *method* and $U, V \subseteq \{T_1, \ldots, T_k\}$. Then, the constraints defined earlier in Definition 10 are checked accordingly:

- $T_i \prec T_j$ – in a valid *solution plan*, all *primitive tasks* and empty vertices in a sub-tree with a root vertex T_i must precede (using the function *order*) all *primitive tasks* and empty vertices in a sub-tree with a root vertex T_j,
- $before(p, U)$ – $p \in s_{i-1}$, i is the minimal value of *primitive tasks* and empty vertices (using *order* function) in sub-trees with roots U,
- $after(U, p)$ – $p \in s_i$, i is the maximal value of *primitive tasks* and empty vertices (using *order* function) in sub-trees with roots U,
- $between(U, p, V)$ – p must hold in states that target $after(U, p)$, $before(p, V)$, and in all states in between.

On contrary to previous models and definitions, in this model the tasks and constraints are not erased nor modified from the *task network*, they are only appended. This practice does not create any inconsistent state, because, in the end, all of the constraints in all vertices must be satisfied for a *plan* to be valid.

A solution to the *planning problem* is a *plan* $\pi = (a_1, \ldots, a_k)$ that is *applicable* to the *initial state*. Moreover, all constraints must be satisfied for each vertex in the constraint graph. The scope of constraints located in a vertex has constraints only for the sub-tree of the given vertex. This feature is possible since the constraint graph is directed and acyclic.

The constraint graph model allows planners to create and verify plans without the demand of new symbols for the representation of *empty methods*. Since the constraints are appended to the model, rather than adjusted, it is possible to spot constraint collisions much earlier. The downside of this model is the graph itself which will need additional space for storing information.

By extending this model and the decomposition definition, it would be possible to use *state-constraints* targeting a decomposing task because the decomposed task is not removed from the graph. For example, $T \rightarrow \varepsilon\ [C], C = \{before(p, T)\}$ would create a new empty vertex $\square$, and later, during the phase of constraint checking, the check would be made in $s_{order(\square)-1}$ (in *the preplan* ϕ).

3.1.4 Index-Based Model

In this subsection, we will inspect a model that will combine ideas of a *task network* defined in Definition 9 and a constraint satisfaction problem [7]. The core idea lies in two functions: $start(t)$ and $end(t)$ with t being a task (*primitive or compound*). These two functions indicate the beginning and the end of the task, or, in other words, the first and the last *primitive task* in *plan* to which the task decomposes. If the task is already *primitive* then $start(t) = end(t)$, and at the same time, this value means the position of the *primitive task* in a valid *plan*. The image of functions *start* and *end* is $\mathbb{N} \cup \{n + 0.5 \mid n \in \mathbb{N}_0\}$. The image of these functions is differentiated between natural numbers which express actions that actually do something (*primitive tasks*) and so-called half-indices which represent *empty methods*. The meaning behind half-indices is that the task that expresses no action or an *empty method* points to a specific place between tasks. This helps the planner to identify the location of *empty methods* and to check all constraints properly. The Index-based model is also allowing real *empty methods* for the cost of extra information that is needed during the planning phase. Like the constraint graph model, the index-based model does not modify the set of constraints, instead, it appends new constraints that need to be satisfied.

Task network, contrary to the constraint graph model, is a pair $\omega = (U, C)$ as in Definition 9. Tasks in U are the inputs to the index functions *start* and *end*. The actual indices of tasks are not known until the *task network* is *primitive* and a valid *plan* is found. Even though the *task network* definition is unchanged, the *method decomposition* differs in this model. Let us be given a *method* $m = T \rightarrow T_1, \ldots, T_k\ [Constr]$ and a *task network* $\omega = (V, C)$ with a *compound task* $T \in V$, the *decomposition* is as follows:

$$\delta(\omega, T, m) = ((V - \{T\}) \cup \{T_1, \ldots, T_k\}, C \cup Constr \cup \{c\}).$$

As we can see, we only remove the task T from the set of tasks V in a *task network* but do not touch any of the existing constraints. New constraints are only added, never modified. For each task $t \in \{T_1, \ldots, T_k\}$ we create new variables $start(t)$ and $end(t)$.
Because we remove the decomposed task T, we need to set up and bind constraints between the removed task and its subtasks (if the method is not empty). That is done with the newly added constraint c in this way:

- $start(T) = min\{start(t') | t' \in \{T_1, \ldots, T_k\}\}$,
- $end(T) = max\{end(t') | t' \in \{T_1, \ldots, T_k\}\}$.

If the *method* is empty (no subtasks), then we add specific constraint $start(T) = end(T)$ and reduce the image of these functions to half-indices, i.e. $\{0.5, 1.5, \ldots\}$.

The meaning of this constraint is that the *empty tasks* (tasks that decompose to nothing via *empty methods*) lie in between tasks that do something (in special cases, before the first *primitive task* or after the last *primitive task* in a *plan*). There is also a possibility that the *primitive task network* would have no tasks at all, in this case, all tasks figuring in a planning phase would point to index 0.5 which indicates the state before the *application* of the first *primitive task* which is equal to the *initial state* in a *planning problem*.

The index-based model allows analogous constraints as Definition 9. Constraint checks are achieved via indices in the following way:

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

where $start(U) = min\{start(t) \mid t \in U\}$ and $end(U) = max\{end(t) \mid t \in U\}$.

Let us be given a *primitive task network* $\omega = (U, C)$, a *plan* $\pi = (a_1, \ldots, a_k)$ (with the associated sequence of intermediate states $(s_0, \ldots, s_k)$) is the *solution* to the *planning problem* if the *task network* was decomposed from the *initial task network* using *methods* from the *planning domain* and all constraints in C are satisfied:

- $before(p, I \in \{1, \ldots, k\})$: $p \in s_{I-1}$,
- $after(I \in \{1, \ldots, k\}, p)$: $p \in s_I$,
- $between(I, p, J)$: $(\forall f)$: $I \leq f < J$: $p \in s_f$.

Another constraints for *the solution* are:

- $(\forall t) : 0.5 \leq start(t) \leq 0.5 + k$, where k is the length of *a plan*
- $(\forall t) : 0.5 \leq end(t) \leq 0.5 + k$, where k is the length of *a plan*.

This way, we make sure that the images of functions *start*, *end* are not widespread more than needed.

All empty tasks are mapped to correct half-indices. At the same time the constraints semantics work properly. Multiple empty tasks can have the same values of *start* and *end*. Suppose a task $t1$ with function values $start(t1) = end(t1) = 3.5$ and a task $t2$ with identical function values. Due to ceiling ($\lceil\ \rceil$) and floor ($\lfloor\ \rfloor$) functions there is no collision in *ordering-constraints*: $t1 \prec t2$: $\lfloor 3.5 \rfloor < \lceil 3.5 \rceil$.
State conditions are also checked correctly with the *empty tasks*. A task $t1$ with function values $start(t1) = end(t1) = 3.5$ is placed correctly between the third and fourth action. In this example, both $before(p, \{t1\})$ and $after(\{t1\}, p)$ are checked in the same state s_3 because $before(p, \lceil 3.5 \rceil)$: $s_{\lceil 3.5 \rceil - 1}$ and $after(\lfloor 3.5 \rfloor, p)$: $s_{\lfloor 3.5 \rfloor}$. This behavior makes perfect sense by cause of the fact that *empty tasks* do not change the state in any sense. Indeed, the state to be checked is after the last applied action in a plan before the *empty task*.

Moreover, every *solution plan* $\pi = (a_1, \ldots, a_k)$ regarding the constraints or *a planning domain* must satisfy these properties:

- for each task t: $start(t) \leq end(t)$,
  - that is obvious from the fact that the *start* values are minimized and *end* values are maximized,
- in *totally-ordered domains* each *compound task* T decomposed to nothing holds $start(T) = end(T)$ (which is half-index),
  - task T is decomposed into nothing. That means, T is not decomposed into any of *actions* thus all of the subtasks of T, his "grand-subtasks" and so on must have only half-indices as values of *start*, *end*. With that, $start(T)$, $end(T)$ must have also half-indices,
  - suppose, that there is T such that $start(T) \neq end(T)$ (both half-indices), this would mean that there exist *an action* that is placed in between indices $start(T)$ and $end(T)$ but that contradicts with *total-ordering* and the fact that T is decomposed into nothing (this *action* would be decomposed from T),
- if two *compound tasks* $t1$, $t2$ with $start(t2) < end(t1) < end(t2)$ (and $t1$ is not decomposed from $t2$) exist then *the planning domain* is not *totally-ordered*,
  - tasks in *totally-ordered domains* that are not decomposed from each other (independent tasks) have disjunctive intervals created with *start*, *end*. Two independent *compound tasks* might have identical values of *start* and *end* but only if they point to the half-index. This is the only possible interleaving.
  - this property is possible only in *partially-ordered domains*,
- no two *compound tasks* $t1$, $t2$ can have identical values of *start* or *end* functions (also $start(t1)$ with $end(t2)$) if they both point to the full-index and $t1$ is not decomposed from $t2$ (and vice versa),
  - each *action* has an unique index (position in *the plan*). Having two independent *compound tasks* and *an action*, at most one of those tasks can have this *action* in their decomposition tree. Therefore, at most one of those tasks can use this index,
- if a *compound task* $t1$ has only one subtask $t2$ then $start(t1) = start(t2)$ and $end(t1) = end(t2)$ holds,
  - trivial,
- a half-index $start/end(t) = k + 0.5$, $k \in \mathbb{N}_0$ indicates that exactly k actions (*primitive tasks*) precede this *empty task*.
  - each *action* is associated with a unique full-index.

Many more properties could be found, these are just some of them.

Similarly to the previous model, this model would also allow *empty methods* with *state-constraints* targeting the decomposed task. This constraint would be added to *the task network* and treated as a normal constraint.

3.1.5 Increment-Based Model

For our last example of HTN semantics handling *empty methods*, we will look at a model with a special context. This time, we will only concern totally-ordered (TO) *planning domains*. Moreover, for simplicity, we will omit the *between-constraint* as it can be simulated with *the before-constraint* and *after-constraints* (will be shown in Algorithm 1). Totally-ordered domains have all *methods* totally-ordered, that is: for every *method* $(T \to T_1, \ldots, T_k\ [C]) \in M$: for every $T_i, T_j \in \{T_1, \ldots, T_k\}, i \neq j$: $T_i \prec T_j$ or $T_j \prec T_i$. This special subclass of *planning domains* is more strict about ordering than partially-ordered (PO) *planning domains*, nonetheless, it allows us to operate with *methods* and *state-constraints* beneficially.

Furthermore, in this model, we will introduce the first model transformation techniques but only in the sense of *task decomposition*. More to this topic will be presented in the following chapter.

The increment-based model tries to solve issues of *empty methods* caused by Definition 10 directly. The semantics of this model are somewhat special and differ from the previous semantics. In this model, not only the *method* selection matters but also the order of the *method* (especially *empty methods*).

The main problem with the Definition 10 is the *task decomposition* using an *empty method*. In this case, all constraints containing a decomposed task are removed, and new modified constraints should be added but because we do not have any subtasks, we cannot say with certainty the semantics of the result *task network*. To solve this problem, we need to deal with *ordering-constraints* and *state-constraints*.

Now we will redefine *task decomposition* that will allow usage of *empty methods*. As was said earlier, we are only concerned about totally-ordered *planning domains*. Having a *method* $m = T \to T_1, \ldots, T_k\ [Constr]$, $k \geq 0$, *a task network* $\omega = (U, C)$ with a *compound task* $T \in U$, the decomposition δ is defined followingly:

$$\delta(\omega, T, m) = ((U - \{T\}) \cup \{T_1, \ldots, T_k\}, C' \cup Constr),$$

where C' is a modification of C:

- replace each *ordering-constraint* containing T with new ones containing $\{T_1, \ldots, T_k\}$, i.e., $\forall i \in \{1, \ldots, k\} : T \prec x$ is replaced with $T_i \prec x$ and $x \prec T$ is replaced with $x \prec T_i$. If $k = 0$, then we just remove all *ordering-constraints* containing T,
- replace each *before-, after-constraint* containing T with new ones containing $\{T_1, \ldots, T_k\}$. For example, we would replace *before-constraint* $before(p, V)$ with $before(p, (V - \{T\}) \cup \{T_1, \ldots, T_k\})$. If $k = 0$, then we have two distinct situations. Either we remove one of the tasks or the last task in the set. Now, let's break down different outcomes:
  - $before(p, V), T \in V, |V| > 1$: we only remove the task T and replace $before(p, V)$ with $before(p, V - \{T\})$,
  - $before(p, V), T \in V, |V| = 1$: we find strict total order on the set U (tasks of the *task network* ω) $T_i \prec T_j \prec \cdots \prec T_k$, and locate the index d of the decomposed task T. After that, we include a new constraint

before$(p, \{T_{d+1}\})$ into the set of constraints and remove the former constraint *before*(p, V). If T_{d+1} does not exist then we create a new constraint *after*$(\{T_{d-1}\}, p)$. If T_{d-1} also does not exist then T is the only task in ω, we can remove the task from the *task network* but the planner needs to check if the *initial state* s_0 contains required *propositional symbol* p,

- *after*$(V, p), T \in V, |V| > 1$: we only remove the task T and replace *after*(V, p) with *after*$(V - \{T\}, p)$,
- *after*$(V, p), T \in V, |V| = 1$: we find strict total order on the set U $T_i \prec T_j \prec \cdots \prec T_k$, and locate the index d of the decomposed task T. After that, we include a new constraint *after*$(\{T_{d-1}\}, p)$ and remove the former *after*(V, p) constraint. If T_{d-1} does not exist, we include *before*$(p, \{T_{d+1}\})$. If T_{d+1} also does not exist then T is the only task in ω, we can remove the task from the *task network* but the planner needs to check if the *initial state* s_0 contains required *propositional symbol* p.

One might ask, why do *state-constraints* concern sets of tasks rather than just a single task? In *the totally-ordered domains* a single task is sufficient enough. The explanation is that there might be multiple orderings of *the methods* and each of the orderings might give a different result. **This semantics acts differently from the previous models because here we prioritize *primitive tasks* to empty tasks**. Different sequences of *empty method* decomposition shift the state to be checked based on the remaining tasks of *a state-constraint*. If there are no tasks left then we shift the constraint to the next (or previous) task. We can see, that with the different sequences of the task decomposition constraint might end up attached to different tasks.

Example 7. *In this example, we will show how the order of* methods *might have different resulting* task networks. *Suppose, we have* a task network $\omega = (\{T_1, T_2, T_3\}, \{T_1 \prec T_2 \prec T_3, before(p, \{T_1, T_3\})\})$ *and two* empty methods $m_1 = T_1 \rightarrow \varepsilon\ [\{\}]$, $m_2 = T_3 \rightarrow \varepsilon\ [\{\}]$.

If we start with $m1$*:* $(\{T_1, T_2, T_3\}, \{T_1 \prec T_2 \prec T_3, before(p, \{T_1, T_3\})\}) \rightarrow (\{T_2, T_3\}, \{T_2 \prec T_3, before(p, \{T_3\})\}) \rightarrow (\{T_2\}, \{after(p, \{T_2\})\})$.
If we start with $m2$*:* $(\{T_1, T_2, T_3\}, \{T_1 \prec T_2 \prec T_3, before(p, \{T_1, T_3\})\}) \rightarrow (\{T_1, T_2\}, \{T_1 \prec T_2, before(p, \{T_1\})\}) \rightarrow (\{T_2\}, \{before(p, \{T_2\})\})$.

It would be also possible to match previous models by allowing only a single task in *state-constraints*. This way, each *empty method* would shift the constraint to the adjacent task. This behavior is equivalent to the previous models.

Now, we will discuss constraint modification. First of all, we will look at the *ordering-constraint*. A *compound task* that is decomposed with an *empty method* is not relevant from the perspective of ordering as this task does not do anything. Hypothetically, between each two actions there is an infinite amount of virtual *empty tasks*, none of them is relevant to the *task network* or the resulting *plan*. For this reason, all *ordering-constraints* containing the decomposed empty task can be removed.

On the other hand, *state-constraints* cannot be removed from the *task network* because they are telling the planner that they need to be checked at some point in the planning phase. Having *an empty method* that decomposes a *compound task* T, all *state-constrains* containing T needs to be modified as this task will be deleted from the *task network*. With our model, we only remove the task from all constraints if the set of tasks in a particular constraint has more than one element. It could also be possible to modify all constraints right away but, for convenience, we have decided to do it this way. If the decomposed task is the last one then we have to modify all constraints holding T to prevent undefined behavior. Since the task is empty it will produce no actions, meaning that there will be no transition between states. The state before the *compound task* T is identical to the state after the T. For this reason, it is possible to easily modify all existing *state-constraints* containing T. The key for the constraint modification is the fact that the *planning domain* is totally-ordered. At any point of planning, we can construct a strict total ordering of the tasks and shift the constraints to the next or the previous task in the ordering.

With the increment-based model, we have finished our list of HTN semantics that solve the problem of *empty methods*. The first two semantics do not allow usage of real *empty methods* meanwhile the last three allow them. Different semantics might be beneficial in different situations, depending on the *planning domain*. Some future software for hierarchical planning might even combine various features from the semantics presented, but that is left for future work.

4 Transformations of HTN Models

After an enumeration of various definitions and semantics, we will finally start discussing the transformation of different HTN models. This chapter will exhibit varying transformations along divergent contexts and preconditions. For example, HTN models might be partially-ordered, totally-ordered, with or without *goal tasks*, having a different set of allowed *state-constraints*.

4.1 Normal Forms

The first section of this chapter is inspired by the *Chomsky normal form* (ChNF) [1]. A context-free grammar (*CFG*) is in ChNF if all of its production rules are of the form: $A \to BC$, $A \to a$, $S \to \varepsilon$ (in this case S cannot be on the right side of a production rule) where A, B, C are nonterminal symbols, a is a terminal symbol, S is a starting nonterminal symbol, and ε denotes an empty symbol. This "nice" form of CF grammar allows us to prove important theorems about CF languages more easily. Similarly, we want to have "nice" forms of *planning domains* without *empty methods* which might speed up algorithms for planning, or plan verification. Also, these forms are useful for proving HTN theorems [3] [4].

Definition 14. *[3] A* hierarchical planning problem $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *is said to be in* $NF_{\geq 2}$ *if all methods are of the form:* $T \to T_1, \ldots, T_k$ $[C], k \geq 2$*;* $T \to a$ $[C]$ *where* $T, T_1, \ldots, T_k$ *are* compound tasks*, and* a *is a* primitive task (action).

Definition 15. *A* hierarchical planning problem $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *is said to be in* HTN-ChNF *if all of its methods are of the form:* $T \to T_1, T_2$ $[C]$*;* $T \to a$ $[C]$ *where* T, T_1, T_2 *are* compound tasks*, and* a *is a* primitive task*.*

In Definitions 14, 15 we exclude the possibility of having empty *plans* but this decision can be interchanged at any time. Moreover, these definitions do not allow *empty methods*.

Definition 16. *We say that two* planning problems $\mathcal{P}'$, $\mathcal{P}''$ *are equivalent if the set of solutions of* $\mathcal{P}'$ *is equal to the set of solutions of* $\mathcal{P}''$*.*

Definition 17. A method $T \to T_1, \ldots, T_k$ $[C]$ *has* linear ordering-constraints C *if subtasks can be split into disjunctive sets (set partition)* $S_1, \ldots, S_k, k \geq 1$ *such that all* ordering-constraints *between tasks are within the same set, and all tasks within one set* S_i *are linearly ordered (if any two tasks aren't comparable within a set directly then we apply a transitive closure). Tasks not contained in any of* ordering-constraints *are in sets of one element.*

Definition 18. *A* hierarchical planning problem $\mathcal{P} = (s_0, \omega, \mathcal{D})$ *is said to be in* HTN-GNF [10] *if it is TO and all of its methods are of the form:* $T \to a, T_1, T_2, \ldots, T_i$ $[C]$*,* $i \geq 0$*; where* T, T_i *are* compound tasks*, and* a *is a* primitive task*.*

Theorem 5. *Every* planning problem $\mathcal{P}$ *(partially-, totally-ordered) that does not generate an empty* plan *as a* solution*, without* state-constraints *(before, after, between) can be transformed into* an equivalent planning problem *in* $NF_{\geq 2}$.

Proof. The proof follows the ideas of the transformation of CFG into ChNF [3]. First of all, we need to get rid of *empty methods*. For that, we need to find all *nullable compound tasks*. *A compound task* T is *nullable* if *a method* $T \to \varepsilon\ [\{\}]$ exists, or if there is a sequence of *decomposition* such that $T \to \cdots \to \varepsilon\ [C]$. *Nullable compound tasks* can be found followingly: for each *method* $T \to \varepsilon\ [\{\}]$, T is *nullable*, and for each *method* $T \to T_1, \ldots, T_k\ [C],\ k \geq 1$ where $T_1, \ldots, T_k$ are *nullable*, T is also *nullable* (*actions* are never *nullable*). Having all *nullable tasks*, we can remove *empty methods* from the domain. After that, for each *method* $T \to T_1, \ldots, T_k\ [C]$ holding $i \geq 1$ *nullable tasks* as subtasks, we create 2^i new methods with/without *nullable tasks*, one special case is if $i = k$ then we do not create a new *empty method* $T \to \varepsilon\ [C]$. All *ordering-constraints* containing removed tasks are also removed in each new method.

The next step is to remove *methods* of a form $T \to T_1\ [\{\}]$ where T, T_1 are *compound tasks*. These *methods* will be called - *unit methods*. For that, we need to find all *unit pairs*. *A unit pair* (T_1, T_2) $T_1 \neq T_2$ is a pair of *compound tasks* such that T_2 can be decomposed from T_1 only with the usage of *unit methods*, i.e., $T_1 \to \cdots \to T_2\ [C]$. After that, for each *unit pair* (T_1, T_2) and for each non-*unit-method* $T_2 \to T_i, \ldots, T_j\ [C_2], i < j$, we create a new method: $T_1 \to T_i, \ldots, T_j\ [C_2], i < j$ (*ordering-constraints* are inherited from the second *method*). Doing so, we can delete all *unit-methods* from the *planning problem* without the modification of *solutions*.

For each *action* a we produce a new *compound task* T_a, a new *method* $T_a \to a\ [\{\}]$, and for each *method* $T \to T_1, \ldots, T_k\ [C]$ having a as a *subtask* we delete a from the *subtasks* and include T_a instead. All *ordering-constraints* having a are interchanged with ones containing T_a.

In this proof, we deleted all *empty methods*, *unit methods*, and modified *methods* so they have single *action* or *compound tasks* as *subtasks*. Thus, all *methods* left have a form: $T \to T_1, \ldots, T_k\ [C], k \geq 2$; $T \to a\ [\{\}]$ where $T, T_1, \ldots, T_k$ are *compound tasks*, and a is *action*. In every step, we did not create nor remove any potential *solution*. □

Theorem 6. *Every* planning problem $\mathcal{P}$ *in* $NF_{\geq 2}$*, with all* methods *containing only* linear ordering-constraints*, without* state-constraints *(before, after, between) can be transformed into* an equivalent planning problem *in* HTN-ChNF.

Proof. In this proof, we will need to divide the large *methods* into smaller ones. Because all tasks are linearly ordered within the divided sets we know that there cannot be "ordering cycles", also we know that at every time in every task set, there is the first task, with respect to *ordering-constraints*. A key feature of such constraints is that the tasks in sets are mutually independent because, from the definition, no *ordering-constraints* exist between tasks from different sets.

Let us have *a method* $T \to T_1, \ldots, T_k\ [C], k \geq 3$, and a set partition of tasks $S_1, \ldots, S_k, k \geq 1$ by the Definition 17. Now, we will transform the input *planning problem* $\mathcal{P}$ into the result *HTN-ChNF planning problem* in two steps. First, we will add new *compound tasks* $S_1, \ldots, S_k$ (same names as set partitions) and $k - 2$ C_i *compound tasks* to the *planning problem*. After that, we will add new *methods*:

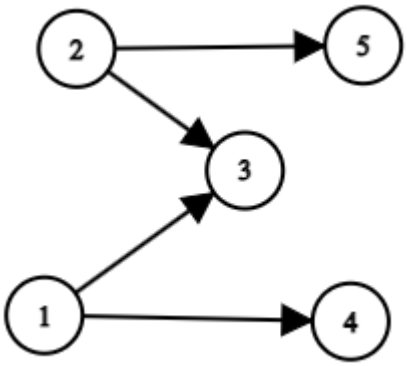

Figure 4.1 Unbinarizable graph

$T \to S_1, C_1$ [{}]; $C_1 \to S_2, C_2$ [{}]; ...; $C_{k-2} \to S_{k-1}, S_k$ [{}]. Newly created methods have an empty set of constraints because, as was mentioned, tasks from different partitions do not have any ordering relations between them. Therefore, their subtasks can interleave without any restrictions.

One special case that might occur is if the set of *subtasks* is fully linearly ordered then there will be only one partition S_1. This situation can be handled easily. We will start the process of binarization right away. Similarly to the previous process, we will create $k-2$ X_i *compound tasks*. After that, we will append new *methods*: $T \to T_1, X_1$ [$\{T_1 \prec X_1\}$]; $X_1 \to T_2, X_2$ [$\{T_2 \prec X_2\}$]; ...; $X_{k-2} \to T_{k-1}, T_k$ [$\{T_{k-1} \prec T_k\}$] where T_1 is the first element with respect to *ordering-constraints*, T_2 is the second element and so on.

In the case of multiple sets $S_1, S_2, \ldots$, we repeat the process above for each *compound task* S_i and for each partition S_i independently. Doing so, we can delete the former *method*. We iterate this process for all *methods* with at least 3 tasks as subtasks. □

Example 8. *If we want to binarize* a planning domain *then we can use different types of* ordering-constraints*.* Method's *subtasks are binarizable if they can be split into two disjunctive sets of subtasks such that all subtasks in one set precede all subtasks in the second set, or if all subtasks in the first set have none* order-constraints *with the second set of subtasks.* A method *is binarizable if this process of splitting into two sets can be done repeatedly until all sets have a single task as an element.*

*Using graphs (*DAG*) terminology where tasks are nodes and an arc* (a, b) *exists if an* order-constraint $a \prec b$ *exists, we can say that the graph can be split if it contains* ≥ 2 *components, or a vertex* u *exists such that in every topological sorting* u *has the same position. If such a vertex exists then we know that all vertices with a higher number must come after* u *and other vertices before. Thus we have 2 disjunctive sets of vertices and* u *can be added to either of these two sets. A graph is binarizable if this process can be repeated until only single vertex is left.*

Figure 4.1 shows us the smallest DAG *that is not binarizable. Now we understand the reason for our choice of the* linear ordering-constraints *in which we can always find the first or the last element.*

Theorem 7. *Theorem 5 transforms* a planning problem $\mathcal{P}$ *with all* methods *having*

linear ordering-constraints *into* a planning problem *in* $NF_{\geq 2}$ *with all* methods *having* linear ordering-constraints.

Proof. Theorem 5 includes two main parts: removing all *empty methods* and *unit-methods*. The elimination of *unit-methods* does not have any effect on ordering because we need at least two tasks for *an ordering-constraint*. We create new *methods* without *nullable* tasks which also deletes constraints containing these tasks. But because the ordering is linear even after the deletion of some elements (transitive closure from the definition) we do not create *methods* that are not *linear ordering-constraints*. □

4.2 Totally-Ordered Transformations

Before we start with the transformation of *totally-ordered planning problems* we need to tidy up *state-constraints*. So far, *state-constraints* are defined on sets of subtasks, i.e., *before*(p, U), *after*(U, p), *between*(U, p, V) with U, V denoting subsets of subtasks, and p denotes *a propositional symbol*. *TO planning problems* allow us to remove unnecessary tasks from sets U, V leaving us with only one subtask. Hence, we abuse the notation to let *before*(p, T) denote *before*$(p, \{T\})$ with T being *a method's* subtask. Similarly, we will use this notation for *after* and *between* constraints.

Theorem 8. *For each* TO planning problem *there is* an equivalent planning problem *with all* state-constraints *operating with tasks instead of sets of sets of tasks.*

Proof. Let us have *a method* $T \to T_1, \ldots, T_k\ [C], k \geq 1$, and a *before*$(p, U) \in C$. This constraint can be transformed to *before*(p, T) such that $T \in U$ and $(\forall F \in (U - \{T\})) : T \prec F$. This can be done analogously for each *after*(U, p) and *between*(U, p, V) constraint. If the result *between*(U, p, V) constraint is of a form *between*(T, p, F) and $F \prec T$ holds then this whole constraint can be removed from the set of constraints as there are none states to test this constraint. □

Now, all *TO* domains will implicitly have only *state-constraints* described in Theorem 8.

In the following parts, we will try to convert general *TO planning problems* into ones of a *HTN-ChNF*. Initially, we will compile away *between* constraints because this type of *state-constraint* spans throughout states which is undesirable. This behavior complicates the conversion because the *HTN-ChNF* has at most two tasks as subtasks. In this case (*HTN-ChNF*), the *between*(T_1, p, T_2) constraint is saying that the check is made before the application of the first *action* to which T_2 decomposes, after the last *action* to which T_1 decomposes, and also all states in between are checked. This means that the test is made in a single state after the last *action* of T_1. This can be easily transformed to a single *before-, after-constraint*. Complications with *between-constraints* begin if we have at least three *TO* subtasks $T_1 \prec T_2 \prec T_3$ and a *between*(T_1, p, T_3) constraint. Having this, the check has to be made in (potentially) multiple states depending on a

decomposition tree of *a compound task* T_2. Transformation of this feature becomes difficult if we do not want to lose any information about *a planning domain*.

In [7] [8] it is described how to get rid of *between* constraints from *TO planning problems* by adding a series of *before* constraints. This conversion is false as it does not concern all states between subtasks but only states before the first *action* to which potential *compound tasks* decompose. Compiling away *between* constraints is more complicated but not impossible. The Algorithm 1 below shows how to do it properly.

In the following Algorithm 1, we will create new *compound tasks* that are derived from the existing ones. Let us have *a compound task* T. A new *compound task* $T_{\{p,q\}}$ (same name but with a suffix) denotes that in every possible decomposition tree starting from $T_{\{p,q\}}$ all *methods* do not contain any *between* constraints, and every *primitive task network* derived from the $T_{\{p,q\}}$ holds *propositional symbols* $\{p, q\}$ in every state. These new *compound task* will be replaced in *methods* containing *between* constraints, and as a result, it will allow us to remove these constraints. Also, we will use the notation $before(Q, T)$ with T being a task and $Q \subseteq P$ being a subset of *propositional symbols*. $before(Q, T)$ is just an abbreviation for a $before(q_1, T), before(q_2, T), \ldots$ where $q_i \in Q$.

Theorem 9. *For each* TO planning problem *there is* an equivalent planning problem *without any* between-constraint.

Proof. First of all, we will compile away *between-constraints* of the neighboring tasks, i.e., tasks $T_1 \prec T_2$ such that no task T_i exist with $T_1 \prec T_i$ and $T_i \prec T_2$. As was said before, these constraints can be interchanged to a single *before-constraint* ($between(T_1, p, T_2) = before(p, T_2)$). *Between-constraints* of a form $between(T_i, p, T_i)$ or $between(T_j, p, T_i)$ with $T_i \prec T_j$ are pointless and can be removed without any loss of information.

The idea of the Algorithm 1 is to remove *between-constraints method* by *method* with a procedure that might remind us of Depth-first search (DFS). Suppose a method M includes at least one *between-constraint*. In every valid *solution* of the *planning problem* $\mathcal{P}$ that employs M during the planning phase, there must exist a consecutive *sub-plan* that is decomposed from the compound(M) and contains the *propositional symbol* specified in the constraint.

If we want to remove $between(T, p, F)$ constraint then we must ensure that all decompositions (even those with cycles) of all *compound tasks* between T and F (with respect to *TO*) hold *propositional symbol* p.

An algorithm is split into two distinct parts: the while-loop (3) in the `Main` function and a recursive function `SearchCompoundTask` (23). The main cycle removes all *between-constraints* from a single *method* per iteration meanwhile not creating any new *between-constraints*. All *compound tasks* that are mentioned in *between-constraints* are substituted with new *compound tasks* that are stored in an auxiliary set `NewCT` (2). If a substituted *compound task* is already stored in `NewCT` then we do not have to recursively search this task with `SearchCompoundTask` function (17, 38). On the contrary, if the task is not searched then we need to create new *methods* so that tasks from other *methods* cannot be decomposed via *methods* designed for the substituted tasks (7, 33).

The recursive procedure `SearchCompoundTask` looks for all *methods* that decompose the input *compound tasks* and append constraints so that the *propositional*

Algorithm 1 TO into TO without between-constraints

1: **procedure** MAIN(*TO planning problem*: $\mathcal{P} = (s_0, \omega, \mathcal{D})$)
2: $NewCT \leftarrow \{\}$ ▷ New *Compound Tasks*
3: **while** $\mathcal{D}$ contains *a method* m with *a between-constraint* **do**
4: **for** each *compound task* CT in $subtasks(m)$ that is part of ≥ 1 *between-constraints* **do**
5: $PropSymbols \leftarrow \{p : between(T, p, F)\ and\ T \prec t \prec F\}$
6: **if** $CT_{PropSymbols}$ not in $NewCT$ **then**
7: find *methods* m' with $CT = compound(m')$ and create new $M = (CT_{PropSymbols}, subtasks(m'), constraints(m'))$
8: **end if**
9: swap all occurrences of CT with $CT_{PropSymbols}$ in *a method* m
10: **end for**
11: **for** each task t in $subtasks(m)$ **do**
12: $PropSymbols \leftarrow \{p : between(T, p, F)\ and\ T \prec t \prec F\}$
13: ▷ if $PropSymbols = \emptyset$ then constraints are always satisfied
14: add $before(PropSymbols, t)$, $after(t, PropSymbols)$ to m
15: **end for**
16: remove all *between-constraints* from $constraints(m)$
17: **for** each $CT_{symbols}$ in $subtasks(m)$ ($CT_{symbols}$ not in $NewCT$) **do**
18: $NewCT \leftarrow NewCT \cup CT_{symbols}$
19: SearchCompoundTask($CT_{symbols}$)
20: **end for**
21: **end while**
22: **end procedure**
23: **procedure** SEARCHCOMPOUNDTASK(*Compound task*: $CT_{InputSymbols}$)
24: **for** each *method* m with $CT_{InputSymbols} = compound(m)$ **do**
25: **for** each task t in $subtasks(m)$ **do**
26: $PropSymbols \leftarrow \{p : between(T, p, F)\ and\ T \prec t \prec F\}$
27: add $before(PropSymbols \cup InputSymbols, t)$ to m
28: add $after(t, PropSymbols \cup InputSymbols)$ to m
29: **end for**
30: **for** each *compound task* CT in $subtasks(m)$ **do**
31: $PropSymbols \leftarrow \{p : between(K, p, L)\ and\ K \prec T \prec L\}$
32: **if** $CT_{PropSymbols \cup InputSymbols}$ not in $NewCT$ **then**
33: find *methods* m' with $CT = compound(m')$ and create new $M = (CT_{PropSymbols \cup InputSymbols}, subtasks(m'), constraints(m'))$
34: **end if**
35: swap all occurrences of CT with $CT_{PropSymbols \cup InputSymbols}$ in m
36: **end for**
37: remove all *between-constraints* from $constraints(m)$
38: **for** each $CT_{symbols}$ in $subtasks(m)$ ($CT_{symbols}$ not in $NewCT$) **do**
39: $NewCT \leftarrow NewCT \cup CT_{symbols}$
40: SearchCompoundTask($CT_{symbols}$)
41: **end for**
42: **end for**
43: **end procedure**

symbols are checked in all states concerning the subtasks (27, 28). Also, the recursive procedure removes all *between-constraints* during the search (35), so they are not found in the main while-loop.

There is a finite number of *propositional symbols*, *methods*, *compound tasks*, and new *compound tasks* in `NewCT`. Therefore, the algorithm will not fall into an infinite cycle, and gradually all *between-constraints* will be removed. □

Theorem 10. *For each* TO planning problem *that does not generate empty* plan *as a* solution*, there is* an equivalent planning problem *without* empty methods*.*

Proof. First, we will remove all *between-constraints* with the Theorem 9.

Similarly to Theorem 5, we will need to find all *nullable tasks* in *a planning problem* and create new *methods* without these tasks. The difficult part is handling the *before-, after-constraints* which need to be checked eventually. *A nullable task* may have many different series of decompositions that lead to an empty set of tasks. Each of these decompositions may have different constraints that need to be checked. For this purpose, we will use a function $Nullifies(T)$ that accepts a *nullable compound task* and returns a set of sets of *propositional symbols*. Each element of the function's output represents one possible way of how we can decompose this *compound task* and which *propositional symbols* need to be checked.

We will find all values of the function $Nullifies$ in the following inductive manner. **Base:** for each $T \to \varepsilon\ [\{\}] : Nullifies(T) = \{\{\}\}$. **Induction:** having $T \to T_1, \ldots, T_k\ [C],\ k \geq 1$ with subtasks containing only *nullable compound tasks* we extend $Nullifies(T) = Nullifies(T)\ \cup\ \{\{$propositional symbols mentioned in $C\}$ $\cup\, propSym_1 \cup \cdots \cup propSym_k\}$ where $propSym_i \in Nullifies(T_i)$. The induction part is repeated until there is a new unique combination of *propositional symbols* that is not mentioned in $Nullifies(T)$ with some *nullable compound task* T and some *method* $T \to T_1, \ldots, T_k\ [C],\ k \geq 1$.

It can be easily seen that T is *nullable* if and only if $Nullifies(T) \neq \emptyset$ (after we have found all values). *The compound task* in *the initial task network* is never *nullable* as stated in the theorem. This feature will be used in the second part of the *empty method* elimination.

At this point, all *empty methods* can be removed from the *planning domain*.

Now, for the final part, we will append new *methods* with/without *nullable tasks* to the *planning problem*, similarly to Theorem 5. Suppose *a method* contains *i nullable tasks* in *subtasks*. Without loss of generality, *nullable tasks* will be $T_1, \ldots, T_i$. There will be $(|Nullifies(T_1)| + 1) * \cdots * (|Nullifies(T_i)| + 1)$ new *methods*, some of them might be identical. For each such combination, we will create a new *method* without (some) *nullable tasks*. *Ordering-constraints* mentioning deleted tasks are removed from the set of constraints. If we want to remove *a nullable task* T_j then we need to find the closest task that is not removed from the *method* and shift *state-constraints*. If the non-removed task T_k holds $k < j$ then we create a new constraint: $after(T_k, PropSymbols)$, $PropSymbols \in Nullifies(T_j)$. On the other hand, if $k > j$ then we append $before(PropSymbols, T_k)$ to the set of *method's* constraints. It might happen that all $subtasks(M)$ of *a method* M are *nullable*, in this case, we do not create new *empty method* because $compound(M)$ will be removed from *methods* that contain $compound(M)$ as a subtask. As was stated before, *initial task network* is never *nullable*, thus we do not need to handle this awkward situation. □

Theorem 11. *For each* TO planning problem *there is* an equivalent planning problem *without any* after-constraints.

Proof. TO planning problems allow us to shift any *after-constraint* to the succeeding task. So, *an after-constraint* is changed to *before-constraint*. This transformation is valid with only one exception. If *the after-constraint* targets the last task with respect to *TO* then *an after-constraint* cannot be shifted to the following task because such task does not exist in a set of *method's subtasks*. One might think that we can compile this feature away using the cooperation with other *methods*. Still, it can happen that the targeted task will be the last *action* in *the solution* which means that the check will be evaluated in a state after the execution of all *actions* in *the plan*. Therefore, in this case, we cannot shift *an after-constraint* so easily. What we can do is to create a new *compound task* T_{empty} with one *empty method*. Then, this task can be inserted at the end of *a method* and problematic *after-constraints* can be shifted to the T_{empty}.

Symmetrically, we can transform all *before-constraints* to the equivalent *after-constraints*. □

Theorem 12. *For each* TO planning problem *that does not generate empty* plan *as a* solution*, there is* an equivalent planning problem *in* HTN-ChNF.

Proof. The proof is akin to proofs of Theorems 5 6 with some extra steps to handle *state-constraints*. At the beginning, we will use Theorem 9 to remove all *between-constraints*. Afterwards, we will use Theorem 10 to remove all *empty methods* which does not create any new *between-constraints*. Now we need to handle *unit methods*. We will find all *unit pairs* which is identical to the proof to the Theorem 5. Analogous to the process of discarding *empty methods unit pairs* might have different decompositions which lead to different sets of constraints. As for *empty methods*, we will find all such combinations of *unit pairs* with sets of constraints with the help of an auxiliary function $Nulifies(T, F)$ with (T, F) $T \neq F$ being an *unit pair*. **Base:** for each *unit method* $T \to F$ $[C]$: $Nullifies(T, F) = \{C\}$ (if there are multiple *unit methods* with identical *compound tasks* but with different constraints then each set of constraints is added separately). **Induction:** having *a unit pair* (T, F) and *a unit method* $F \to H$ $[C]$: $Nullifies(T, H) = Nullifies(T, H) \cup \{constr_t \cup constr_h \mid (\forall constr_t \in Nullifies(T, F))$ *and* $(\forall constr_h \in Nullifies(F, H))\}$. Hence, for each *unit pair* (T, F) and for each *non-unit method* $F \to F_1, \ldots, F_k$ $[C]$ we create new *methods* for each set of constraints in $Nullifies(T, F)$. New *method* is of a form $T \to F_1, \ldots, F_k$ $[C \cup newC]$ where $newC$ are newly produced constraints from $newConstr \in Nullifies(T, F)$. $newC = \{before(p, F_1) \mid before(p, X) \in newConstr\} \cup \{after(F_k, p) \mid after(X, p) \in newConstr\}$. After that, we can remove all *unit methods*.

For each *method* with multiple subtasks with at least one *primitive task* as a subtask, we substitute *a primitive task* a with *a compound task* T_a. All constraints targeting a are interchanged with constraints targeting T_a. We also create a new *method* $T_a \to a$ $[\{\}]$. Now, our *planning problem* is in $\text{NF}_{\geq 2}$. To finish the proof we need to divide long *methods*.

For each *method* $T \to T_1, \ldots, T_k$ $[C]$, $k \geq 3$ we produce $k - 2$ C_i *compound tasks*, and construct new *methods* $T \to T_1, C_1$ $[\{T_1 \prec C_1\} \cup \{$all *state-constraints* targeting $T_1\}]$, $C_1 \to T_2, C_2$ $[\{T_2 \prec C_2\} \cup \{$all *state-constraints* targeting $T_2\}]$,

$\ldots, C_{k-2} \rightarrow T_{k-1}, T_k$ $[\{T_{k-1} \prec T_k\} \cup \{\text{all } \textit{state-constraints} \text{ targeting } T_{k-1}, T_k\}]$. □

Theorem 13. *For each* TO planning problem *in* HTN-ChNF *there is* an equivalent planning problem *in* HTN-GNF.

Proof. We assume that *the planning problem* does not contain any *between-constraints* as they can be easily eliminated 4.2 in *HTN-ChNF*.

The proof follows the [11] with the addition of HTN specifics. Initially, we rename *compound tasks* to $A_1, A_2, \ldots, A_n$ in some ascending order (and substitute these new tasks in constraints). *The compound task* in *the initial task network* is renamed to A_1.

Our next goal is to modify *methods* such that $A_i \rightarrow A_j, \alpha\ [C] \implies i < j$, where A_j is the first *compound task* with respect to ordering and α, is a set of *compound tasks*. For this, we will iterate over A_i starting from $i = 1$ and proceeding to $i = n$. Having $A_i \rightarrow A_j, \alpha\ [C]$, we might have two scenarios that need adjustment:

1. If $i > j$ then we generate new *methods* substituting A_j with $\textit{subtasks}(M)$ such that $\textit{compound}(M) = A_j$ for *methods* M. Constraints of these new *methods* are unioned with the old one ($\textit{constraints}(M) \cup C$). After we generate new *methods* we can remove the unsuitable one.

2. (a) If $i = j$ then we have encountered the incompatible left recursion. To remove that, we need to handle constraints targeting A_j. Now we need to find all *methods* with $\textit{compound}(M) = A_i$ and separate them into two groups (with/without left recursion):

$$A_i \rightarrow A_i, \alpha_1\ [C_{\alpha_1}] \mid \ldots \mid A_i, \alpha_m\ [C_{\alpha_m}],$$

$$A_i \rightarrow \beta_1\ [C_{\beta_1}] \mid \ldots \mid \beta_p\ [C_{\beta_p}].$$

For each $A_i \rightarrow A_i, \alpha\ [C]$ we remove all *after-constraints* targeting A_i and append new *before-constraints* targeting the first task in α (same state in *a plan*). This can always be done as we have *HTN-ChNF* which contains two *compound tasks* in subtasks (if the first one is *compound*). To handle *before-constraints* targeting A_i, we need to find all possible outcomes of all recursions. We will construct a

$$\Omega = \{\{p \in P \mid \textit{before}(p, A_i) \in C\} \mid (A_i \rightarrow A_i, \alpha\ [C]) \in M\}\ (|\Omega| = m).$$

In other words – a set of sets with *propositional symbols* where each element is a union of *propositional symbols* found in *a before-constraints* targeting the first task in *a method* with a left recursion of A_i.

With these two groups, we eliminate left recursion by introducing new *compound tasks* Z_Q^R with $R \subseteq Q \subseteq \{1, \ldots, |\Omega|\}$ (Q is a set of possible decompositions, R is a set of mandatory decompositions that have not been used yet) and a bijective function $f : \{1, \ldots, |\Omega|\} \rightarrow \Omega$

(s.t. $f(j) = \{p \in P \mid before(p, A_i) \in C_{\alpha_j}\}$). We will substitute these *methods* with:

$$A_i \to \beta_i\ [C_{\beta_i}] \mid \beta_i, Z_Q^Q\ [C_{\beta_i} \cup \{before(f(Q), \beta_i) \mid Q \subseteq \{1, \ldots, |\Omega|\}\} \cup \{\beta_i \prec Z_Q^Q\}],$$

$$Z_Q^R \to \alpha_{q \in Q\ (R=\emptyset)} \mid \alpha_{q \in Q}, Z_Q^R \mid \alpha_{r \in R}, Z_Q^{R-\{r\}}.$$

(b) Alternatively [12], we could remove left recursion differently with lesser number of *methods*. With this approach, it could be possible to create *task networks* with more *state-constraints* than needed, but it will not change the set of possible solutions of the domain. As in the previous approach, we will construct Ω and f. New *methods* will look as follows:

$$A_i \to \beta_i\ [C_{\beta_i}] \mid \beta_i, Z_Q\ [C_{\beta_i} \cup \{before(f(Q), \beta_i) \mid Q \subseteq \{1, \ldots, |\Omega|\}\} \cup \{\beta_i \prec Z_Q\}],$$

$$Z_Q \to\ \alpha_j, Z_Q\ (\{p \in P \mid before(p, A_i) \in C_{\alpha_j}\} \subseteq f(Q)),$$

$$Z_Q \to \alpha_j\ (\{p \in P \mid before(p, A_i) \in C_{\alpha_j}\} \subseteq f(Q)).$$

The next step is to make sure that $A_i \to pt, \alpha\ [C]$ with *a primitive task pt* holds for all *methods* with $compound(M) = A_i$. For this, we will start from $i = n-1$ and move on to $i = 1$. For each i we will find all unsuitable *methods* $A_i \to A_j, \alpha\ [C]$. These *methods* will be substituted with $subtasks(M)$ and $constraint(M)$ as was done earlier in this proof.

The same procedure will be carried out to the new *methods*, where $compound(M) = Z_Q^R$. □

So far, we have only discussed *methods* with *constraints* that can target subtasks. This restriction is placed in Definition 10. Besides that, there are HTN formalisms that allow *state-constraints* targeting a *compound task* that is being decomposed. For example $T \to T_1, T_2\ [T_1 \prec T_2, before(p, T)]$ or $T \to \varepsilon\ [after(T, q)]$. The following transformation will try to convert these *methods* into *methods* that satisfy our definitions.

Theorem 14. *For each* HTN method *in* TO planning problem *(including* empty methods*) containing* after-, before-constraints *targeting a decomposed* task*, there is an equivalent transformation that removes these* constraints*.*

Proof. Suppose we have *a method* $T \to T_1, T_2, \ldots, T_k\ [C],\ k \geq 0$ with (for example) $before(p, T) \in C$. Until there are such *constraints*, choose any of them, remove it from the set of *constraints*, create a new unique *compound task* X and a new *empty method* $X \to \varepsilon\ [\{\}]$. If the removed *constraint* was a *before-constraint* then append X to the beginning of *a method* (X proceeds all current *tasks*) and

append a new *before-constraint* $before(p, X)$ to the set of *constraints*. Analogously for the *after-constraints*. Repeat until there are such *constraints*. This procedure works fine for *empty methods*. □

Example of the transformation above:

1. $T \rightarrow T_1$ $[before(p, T), after(T, q)]$
2. $T \rightarrow X_1, T_1$ $[X_1 \prec T_1, before(p, X_1), after(T, q)]$
3. $T \rightarrow X_1, T_1, X_2$ $[X_1 \prec T_1 \prec X_2, before(p, X_1), after(X_2, q)]$

Almost identical transformation can be used for *the partially-ordered planning domains*. Key difference is that *PO methods* allow task interleaving. For this reason, this transformation is limited to *the empty methods* and *unit methods*. Having such *a method*, select all problematic *state-constraints*, remove them from *the method* and append new constraints targeting the only subtask. In case of *an empty method*, also create a new task X and *a new method* $X \rightarrow \varepsilon$ $[C]$.

Conclusion

In summary, the main purpose of this thesis was to provide a brief introduction to the field of planning and to explore some aspects that can be used later in other types of related work. Starting from classical planning which can be expressed in different ways and lasting with hierarchical planning, HTN to be precise. The theory of HTN is not yet unified, therefore we can find different definitions and understandings about this topic in various publications.

One of the goals was to establish proper boundaries through combinations of definitions from varying sources. By doing so, we could describe, compare, and analyze HTN semantics with the subtle goal of handling empty methods that are not handled accurately in plenty of similar papers. Difficulties start to appear if we want to use constraints that are bound to states. In addition, new types of HTN semantics were introduced.

In the last chapter of the thesis, we tried to find transformations of HTN models that might be suitable within some context. Most transformations were inspired by the theory of Automata and Grammars, yet they need a significant amount of extension as HTN models allow partially-ordered domains and state-constraints.

We have shown a few transformations regarding (mostly) TO *planning domains*. These transformations can be used for plan validation and recognition. Future experiments with these normal forms may result in an increased speed of computation. In particular, GNF may be useful, as in every step of a plan validation, one *primitive task / action* is placed.

Bibliography

1. Chytil, M. *Automaty a gramatiky*. SNTL - Státní nakladatelství technické literatury, 1984.
2. Erol, K.; Hendler, J. A.; Nau, D. S. Complexity Results for HTN Planning. *Annals of Mathematics and AI 18(1)*. 1996, pp. 69–93.
3. Höller, D.; Behnke, G.; Bercher, P.; Biundo, S. Language classification of hierarchical planning problems. *ECAI*. 2014, pp. 447–452.
4. Lin, S.; Behnke, G.; Ondrčková, S.; Barták, R.; Bercher, P. On Total-Order HTN Plan Verifcation with Method Preconditions – An Extension of the CYK Parsing Algorithm. *The Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI-23)*. 2023, pp. 12041–12048.
5. Fikes, R. E.; Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *IJCAI 1971*. 1971, pp. 608–620.
6. Ghallab, M.; Nau, D.; Traverso, P. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, 2004. isbn 1-55860-856-7.
7. Ondrčková, S.; Barták, R. On Semantics of Hierarchical Planning Domain Models with Decomposition Constraints and Empty Methods. *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*. 2023, pp. 349–353.
8. Ondrčková, S.; Barták, R. Handling Empty Decomposition Methods in Hierarchical Planning. *FLAIRS*. 2024.
9. Höller, D; Behnke, G; Bercher, P; Biundo, S; Fiorino, H; Pellier, D; Alford, R. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020, vol. 34, pp. 9883–9891.
10. Hopcroft, J. E.; Motwani, R; Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2001.
11. Barták, Roman. *Lecture 09: Pushdown Automata* [`https://web.archive.org/web/20230515195248/https://ktiml.mff.cuni.cz/~bartak/automaty/lectures/lecture09.pdf`]. 2013. Archived version accessed on January 27, 2025.
12. Barták, R. On Removing Left Recursion from Totally-ordered HTN Domain Models. *Submitted to FLAIRS-38*. 2025.

List of Figures

List of Abbreviations

AI – Artificial Intelligence
BFS – Breadth-First Search
CFG – Context-free Grammar
ChNF – Chomsky Normal Form
DAG – Directed Acyclic Graph
DFA – Deterministic Finite Automaton
DFS – Depth-First Search
GNF – Greibach Normal Form
HDDL – An Extension to PDDL for Expressing Hierarchical Planning Problems
HTN – Hierarchical Task Network
LLM – Large Language Models
PO – Partially-ordered
TO – Totally-ordered

A Attachments

A.1 Program Manual

NAME

htn-transformator - a command-line program that demonstrates some of the transformations mentioned in this text.

SYNOPSIS

```
htn-transformator -i INPUTFILE [--between|--empty|--tocnf]
[-o OUTPUTFILE]
```

DESCRIPTION

`htn-transformator` is a utility designed to apply specific transformations to HTN planning domains.

OPTIONS

-i INPUTFILE Specifies the input file containing the HTN data. This option is mandatory.

–**between** Removes all between constraints from the input planning domain.

–**empty** Removes all between constraints and empty methods from the input planning domain.

–**tocnf** Transforms the input planning domain into the HTN-ChNF.

-o OUTPUTFILE Specifies an optional output file to save the result. If this option is omitted, the output is displayed to stdout.

A.2 Input Domain Format

Each planning domain is defined as a list of **methods**. Each method is placed on a separate line and consists of three parts:

1. **Head compound task**
2. **Subtasks**
3. **Constraints**

Syntax:

$$\texttt{HEAD} - - > (\texttt{subtask1}, \texttt{subtask2}, \ldots); [\texttt{constr1}, \texttt{constr2}, \ldots]$$

Rules

- No white spaces are allowed.
- To allow multiple identical tasks, each subtask is indexed: `SUBTASK#index`.
- Compound tasks start with a capital letter.
- Primitive tasks start with a lowercase letter.
- Multiple subtasks with identical name and index are omitted.
- Task names must start with a letter and may include letters and digits.
- All constraints must target subtasks mentioned in the subtasks section.
- There are four types of constraints:
  - Ordering constraint (between two or more tasks)
  - `before`
  - `after`
  - `between`
- Each state constraint contains a single propositional symbol checked during planning.

Comment Syntax

- Lines enclosed in `/*` and `*/` are ignored. These symbols must be at the beginning of a line.
- Lines starting with `#` are ignored.
- Empty lines are skipped by the parser.

```
/*
insideComment
insideComment

insideComment
*/

# ignored
```