Journal of Artificial Intelligence Research 80 (2024) 613-663 Submitted 06/2023; published 06/2024

The Toad System for Totally Ordered HTN Planning

Daniel Höller HOELLER@CS.UNI-SAARLAND.DE
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany

Abstract

We present an approach for translating Totally Ordered Hierarchical Task Network (HTN) planning problems to classical planning problems. While this enables the use of sophisticated classical planning systems to find solutions, we need to overcome the differences in expressiveness of these two planning formalisms. Prior work on this topic did this by translating *bounded* HTN problems. In contrast, we *approximate* them, i.e., we change the problem such that every action sequence that is a solution to the HTN problem is also a solution for the classical problem, but the latter might have more solutions. To obtain a sound overall approach, we verify solutions returned by the classical planning system to ensure that they are also solutions to the HTN problem.

For translation and approximation, we use techniques introduced to approximate Context-Free Languages by using Finite Automata. We named our system Toad (Totally Ordered HTN Approximation using DFA). For a subset of HTN problems the translation is even possible without approximation. Whether or not it is necessary is decided based on the property of *self-embedding*, which comes also from the field of formal languages. We investigate the theoretical connection of self-embedding and *tail-recursiveness*, a property from the HTN literature used to identify a subclass of HTN planning problems that can be translated to classical planning, and show that it is more general. To guide the classical planner, we introduce a novel heuristic tailored towards our models.

We evaluate Toad on the benchmark set of the 2020 International Planning Competition. Our evaluation shows that (1) most problems can be translated without approximation and that (2) Toad is competitive with the state of the art in HTN planning.

1. Introduction

Planning is the task of generating a sequence of actions that achieve some objective. It is solved based on a model describing the environment and how to change it. Two well-studied approaches to planning are classical planning and hierarchical planning, in the latter case especially Hierarchical Task Network (HTN) planning.

In classical planning, the environment is usually described by a propositional model. The state of the environment can be changed by a set of actions. Each action comes with a set of preconditions that need to hold to be able to apply it, and a set of state features that are added and deleted by the action. The objective is to find a sequence of actions that is applicable in the current state and that results in a state where certain state features hold.

Models in HTN planning additionally define a grammar-like decomposition structure. Solutions need to be derived via that grammar, so there are two means to restrict the set of solutions: state transition as given before, and the hierarchy. The motivations to use hierarchical planning are manifold: HTN planning is more expressive than classical plan-

©2024 The Authors. Published by AI Access Foundation under Creative Commons Attribution License CC BY 4.0.

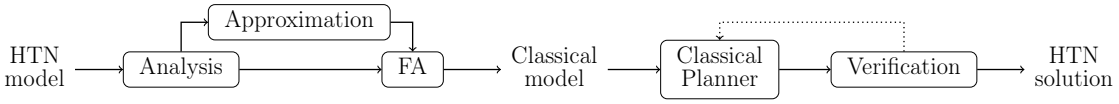

Figure 1: Overview of the Toad approach. First, it is analyzed whether the HTN problem can be translated exactly, or if approximation is needed (see Section 4.1). When approximation is needed, the grammar rules (which in our case are the HTN methods) are transformed such that the set of solutions increases (Section 4.3). Next, a FA is built accepting action sequences derivable via the hierarchy (Section 4.4) and a classical planning problem is encoded capturing the FA and the original state transition (Section 4.5), which is solved using a classical planning system. To return only valid solutions for the original HTN problem, verification is applied as last step. This figure is based on Figure 1 from Höller (2021).

ning (Erol et al., 1996; Höller et al., 2014, 2016), while commonly-used classical planning formalisms can express structures similar to the regular languages, the full HTN formalism can express (non-context-free) context-sensitive structures, and the subclass of totally ordered HTN planning used here can exactly express the context-free languages. It has been shown that HTN planning can express undecidable problems like the grammar intersection problem of context-free languages (see Erol et al., 1996). It has also been argued that hierarchical formalisms sometimes enable a more natural way of modeling a domain, e.g. when it is known which medical treatment will lead to the recovery of a patient without knowing the actual effects of each step (Goldman, 2009). Hierarchical models have also been used to communicate with human users on different levels of abstraction (Köhn et al., 2020; Behnke et al., 2020). Another motivation is to include advice on how to solve a problem into a planning model that could in principle also be modeled using classical models (see e.g. SHOP2 by Nau et al., 2003, and SHOP3 by Goldman & Kuter, 2019).

This leads to a large variety in HTN models: some need the full expressiveness, others use the hierarchy for other reasons; some models include enough advice to use uninformed search (usually depth first search) to find solutions very fast, other models do not include any advice. This variety is also reflected in the benchmark set used in the 2020 International Planning Competition (IPC) (Behnke et al., 2021).

There is also a range of different solving techniques used in HTN planning. Many solvers are based on search (see e.g. Nau et al., 2003; Bit-Monnot et al., 2016; Goldman & Kuter, 2019; Höller et al., 2020; Pellier & Fiorino, 2021; Ramoul et al., 2017; Lesire & Albore, 2021), but there are also translations to propositional logic (see e.g. Behnke et al., 2018, 2019; Schreiber et al., 2019; Schreiber, 2021a, 2021b), or non-hierarchical planning (see e.g. Alford et al., 2009, 2016; Behnke et al., 2022). We provide a detailed discussion of related work in Section 6. A summary can also be found in the survey by Bercher et al. (2019).

In this article, we present a novel translation-based approach for the subclass of *totally ordered* HTN planning. Our approach is based on techniques from formal languages introduced to build a finite automaton (FA) that accepts the language of a context-free grammar (CFG) (see Nederhof, 2000a, 2000b). We use them to create a FA that specifies

which action sequences can result from the HTN decomposition process. Since FAs and CFGs describe different language classes, this is in general not directly possible, of course. However, a direct translation is possible for models that are not *self-embedding* – a property which also comes from the field of formal languages. For models that *are* self-embedding, we apply an approximation introduced by Nederhof, 2000a, 2000b. Nederhof introduces two kinds of approximations: the first one results in an increase of the original solution set (superset approximation), the second one results in a decrease (subset approximation). To get a complete overall approach, we use only superset approximation in this work, i.e., the FA describes a superset of the HTN solutions. We combine this FA with the original actions from the HTN model to produce a classical planning model. To make our approach sound, we verify the solutions returned by the classical planning system using an approach from the literature (Höller et al., 2022). We named our system Toad (Totally Ordered HTN Approximation using DFA). The overall process is illustrated in Figure 1. Our contributions are the following:

- We introduce Toad, a translation-based planner for totally ordered HTN planning.
- We analyse the theoretical connection of self-embedding and a property called *tail-recursiveness*, which was introduced by Alford et al. (2016) as a property identifying a subclass of HTN planning problems that can be translated to classical planning. We show that the property used here is more general, i.e., we show that every totally ordered tail-recursive HTN problem is also non-self-embedding, but not vise versa.
- We analyse the benchmark set from the 2020 International Planning Competition. Interestingly, wide parts of the benchmarks are not self-embedding and can thus be translated exactly.
- To solve the classical planning problems resulting from the translation, we introduce a heuristic tailored towards our problems. It re-uses information from the translation to create a pre-computed heuristic.
- We study the impact of minimization and determinization of the FAs on their size and the resulting planning performance.
- Our evaluation on the 2020 IPC benchmark set shows that our system is competitive with the state of the art in HTN planning.

The basic approach has been presented by Höller (2021). We extend this work along a practical and a theoretical line. On the practical side, optimization for the generated automata as well as the classical heuristic tailored to our problems have not been presented before. Further, the theoretical connection between self-embedding and tail-recursiveness has not been part of the conference publication.

2. Preliminaries

We now introduce the formalisms used throughout the article. We start with a classical model in finite domain representation (FDR) (Helmert, 2006; Bäckström & Nebel, 1995) and combine it with an HTN formalism specialized to totally ordered HTN planning as introduced by Behnke et al. (2018).

2.1 Classical Planning Problems

A classical planning problem is a tuple $\Pi_c = (\mathcal{V}, \mathcal{A}, s_0, s_\star)$. $\mathcal{V}$ is a set[1] of variables, each variable $v \in \mathcal{V}$ has a finite domain $\mathcal{D}_v$. A *partial assignment* is a function mapping a subset of the variables to values of their respective domains. We write $v = d$ to denote that the variable $v \in \mathcal{V}$ has the value $d \in \mathcal{D}_v$. Let s and s' be partial assignments. We use subset notation $s' \subseteq s$ to denote that all variables contained in s' are also assigned in s and have the same values, i.e., $(s'(v) = d) \Rightarrow (s(v) = d)$. We write $vars(s)$ to denote the set of variables set in the partial state s. Let V be a set of variables, we write $s[V]$ to denote the partial state where exactly the variables contained in V have a value assigned. A *state* is a partial assignment on all variables. s_0 is the initial state of the problem. $s_\star$ is a partial state defining which assignments must hold in a state to be a goal state.

$\mathcal{A}$ is a set of actions. There are functions *prec* and *eff* that map each action to partial assignments, the actions' preconditions and effects, respectively. An action $a \in \mathcal{A}$ is applicable in a state s if and only if $prec(a) \subseteq s$. When an applicable action a is applied to a state s, the state $s[\![a]\!]$ resulting from the application is defined as $s[vars(s) \setminus vars(eff(a))] \cup eff(a)$, i.e., the values of variables contained in the effect are set to the value given in the effect, the other values remain unchanged. A sequence of actions $a_1, a_2, \ldots, a_n$ is applicable in a state s_0 when each a_i is applicable in the state s_{i-1} with $s_i = s_{i-1}[\![a_i]\!]$ for $1 \leq i \leq n$. We call s_n the state resulting from the application.

A *solution* (also *plan*) to a classical planning problem is a (possibly empty) sequence of actions that is applicable in s_0 for which the application results in a state s_n that is a *goal state*, i.e., $s_\star \subseteq s_n$.

2.2 Totally Ordered HTN Planning Problems

A totally ordered HTN (TOHTN) planning problem is a tuple $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$. The elements $\mathcal{V}$, $\mathcal{A}$, s_0, and $s_\star$ are defined as given before.

$\mathcal{C}$ is a set of abstract tasks. These tasks cannot be applied directly, they need to be decomposed in a process that – for totally ordered HTN planning – exactly resembles the derivation of words from a context-free grammar. Let $\mathcal{N} = \mathcal{C} \cup \mathcal{A}$ (wlog. we assume that $\mathcal{C} \cap \mathcal{A} = \emptyset$). A task network is an element out of $\mathcal{N}^*$ (where * is the Kleene operator).

The set of methods $\mathcal{M} \subseteq \mathcal{C} \times \mathcal{N}^*$ defines how tasks can be decomposed. A method (c, φ) is applicable to the task $c \in \mathcal{C}$. When it improves readability, we write $c \to \varphi$ instead of (c, φ). When a method (c, φ) is applied to a task network $\omega c \omega'$ with $\omega, \omega' \in \mathcal{N}^*$, the resulting task network is defined as $\omega \varphi \omega'$. We write $\omega \rightsquigarrow \omega'$ to denote that a task network ω can be decomposed into a task network ω' by applying zero or more methods[2]. The *initial task* $c_I \in \mathcal{C}$ is the task the decomposition process starts with.

A task network ω is a solution to a TOHTN planning problem if and only if the following solution criteria hold.

1. $c_I \rightsquigarrow \omega$ – it can be reached by decomposing the initial task (Solution Criterion 1).

1. All sets given here in the definition of classical and HTN planning are finite.
2. We use three different arrow symbols: $\rightsquigarrow$ as defined here; $\to$ as given above to define methods, where $c \to \varphi$ denotes a method decomposing c into φ; and $\mapsto$ to define functions like $prec(a) \mapsto F$, where F is the set of state features returned by the function *prec* when applied to action a.

2. $\omega \in \mathcal{A}^*$ – all contained tasks are primitive (Solution Criterion 2).

3. ω is applicable in the initial state and results in a goal state[3] (Solution Criterion 3).

2.3 From Planning Problems to Formal Languages

The set of solutions to an HTN planning problem is restricted by two mechanisms, which are represented in the solution criteria: Criteria 1 and 2 restrict the set to primitive task sequences resulting from decomposing the initial task. Criterion 3 enforces the state-based transition semantics like present in classical planning.

We can regard the set of action sequences resulting from these two as formal languages, the first one containing sequences in line with Criteria 1 and 2, the second one those in line with Criterion 3. The solutions to the overall problem need to fulfill *all* solution criteria, therefore the set of solutions to the overall problem is formed by the intersection of the two languages (Höller et al., 2014). Since the number of states in a planning problem is finite, the second language is regular (Höller et al., 2016). In totally ordered HTN planning, the definition of decomposition methods and their application (i.e., the "decomposition" itself as defined in Section 2.2) exactly resemble a context-free grammar and the respective derivation of words. As a result, the second language is a context-free language (Höller et al., 2014). Therefore is also the overall "language" (the set of all solutions to the problem) context-free.

Definition 1 (The languages L_h and L_c)**.** *Let $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$ be an HTN planning problem. We define the language L_h as the set of all action sequences that fulfill Criteria 1 and 2; and L_c as the set of all action sequences that fulfill Criterion 3.*

Let L_{Π_h} be the set of solutions to a planning problem Π_h, then $L_{\Pi_h} = L_h \cap L_c$ (Höller et al., 2014).

Definition 2 (The grammar G_h)**.** *Given a planning problem $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$, we define G_h as a tuple $(\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$.*

The set of abstract tasks $\mathcal{C}$ can be regarded a set of non-terminal symbols, the actions $\mathcal{A}$ as terminals, $\mathcal{M}$ as production rules, and c_I as a start symbol. Given the way "decomposition" is defined in Section 2.2, the following theorem holds:

Theorem 1 (G_h defines L_h)**.** *G_h is a context-free grammar defining L_h.*

The grammar is defined for a particular planning problem. However, in the following it will be clear which one we refer to and we omit further annotation to keep notation simple.

3. Running Example

Throughout the article, we illustrate our approach using variations of a simple transport problem as running example. The lifted HDDL (Höller et al., 2020a) domain model is contained in the appendix. Here we present the grounding of a small instance. It has been created using the Panda grounding system (Behnke et al., 2020), which is also used in the preprocessing of our Toad system.

3. In HTN planning, the state-based goal condition is usually empty. When there is one, it can be compiled away by introducing an artificial last task that has the state-based goal definition as its precondition and thus enforces it to hold in the end.

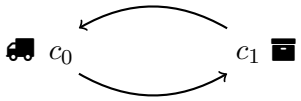

Figure 2: Illustration of the initial state of our running example.

The initial state of the problem instance is shown in Figure 2. A truck t is located at position c_0 and a package at position c_1. There are two variables $l(p)$ and $l(t)$ describing the locations of the package and the truck, respectively. The package might be at one of the two locations or in the truck, i.e., $\mathcal{D}_{l(p)} = \{at(p, c_0),\ at(p, c_1),\ in(p, t)\}$. The truck might be at some position, i.e., $\mathcal{D}_{l(t)} = \{at(t, c_0),\ at(t, c_1)\}$. The package shall be delivered to location c_0. This is ensured by the state-based goal definition, $s_\star = \{l(p) = at(p, c_0)\}$.

3.1 State Transition System

There are actions to move the truck between the two positions, to pick-up and to drop the package at both positions.

$$\begin{aligned}\mathcal{A} = \{&drive(t, c_0, c_1), drive(t, c_1, c_0),\\ &pick\text{-}up(t, c_0, p), pick\text{-}up(t, c_1, p),\\ &drop(t, c_0, p), drop(t, c_1, p)\}\end{aligned}$$

The preconditions and effects of the actions are defined as expected in such a domain:

$$\begin{aligned}prec(drive(t, c_0, c_1)) &\mapsto \{l(t) = at(t, c_0)\}\\ eff(drive(t, c_0, c_1)) &\mapsto \{l(t) = at(t, c_1)\}\\ prec(drive(t, c_1, c_0)) &\mapsto \{l(t) = at(t, c_1)\}\\ eff(drive(t, c_1, c_0)) &\mapsto \{l(t) = at(t, c_0)\}\end{aligned}$$

$$\begin{aligned}prec(pick\text{-}up(t, c_0, p)) &\mapsto \{l(p) = at(p, c_0), l(t) = at(t, c_0)\}\\ eff(pick\text{-}up(t, c_0, p)) &\mapsto \{l(p) = in(p, t)\}\\ prec(pick\text{-}up(t, c_1, p)) &\mapsto \{l(p) = at(p, c_1), l(t) = at(t, c_1)\}\\ eff(pick\text{-}up(t, c_1, p)) &\mapsto \{l(p) = in(p, t)\}\end{aligned}$$

$$\begin{aligned}prec(drop(t, c_0, p)) &\mapsto \{l(t) = at(t, c_0), l(p) = in(p, t)\}\\ eff(drop(t, c_0, p)) &\mapsto \{l(p) = at(p, c_0)\}\\ prec(drop(t, c_1, p)) &\mapsto \{l(t) = at(t, c_1), l(p) = in(p, t)\}\\ eff(drop(t, c_1, p)) &\mapsto \{l(p) = at(p, c_1)\}\end{aligned}$$

3.2 Decomposition Structure

The set of abstract tasks is defined as:

$$\begin{aligned}\mathcal{C} = \{&\textit{logistics-problem}(),\\ &\textit{deliver}(p, c_0), \textit{deliver}(p, c_1),\\ &\textit{get-to-s}(t, c_0), \textit{get-to-s}(t, c_1),\\ &\textit{get-to-d}(t, c_0), \textit{get-to-d}(t, c_1)\}\end{aligned}$$

The decomposition process starts with the abstract task *logistics-problem*(), which describes one transport problem with one or more packages. It can be decomposed using the following set of methods:

$$\begin{aligned}&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_0), \textit{logistics-problem}()\\ &\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_1), \textit{logistics-problem}()\\ &\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_0)\\ &\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_1)\end{aligned}$$

The last two methods decompose the overall logistics problem into the delivery of a single package. There is one deliver task for each combination of package and target location. Since there is only one package and two locations in this instance, there are two such tasks, one to deliver p at location c_0, and one to deliver p at location c_1. The first two methods decompose the *logistics-problem*() task into one of the deliver tasks followed by another *logistics-problem*() task, i.e., it enables the delivery of more than one package. Deliver tasks can be decomposed using one of the following methods:

$$\begin{aligned}&\textit{deliver}(p, c_0) \rightarrow \textit{get-to-s}(t, c_0), \textit{pick-up}(t, c_0, p), \textit{get-to-d}(t, c_0), \textit{drop}(t, c_0, p)\\ &\textit{deliver}(p, c_0) \rightarrow \textit{get-to-s}(t, c_1), \textit{pick-up}(t, c_1, p), \textit{get-to-d}(t, c_0), \textit{drop}(t, c_0, p)\\ &\textit{deliver}(p, c_1) \rightarrow \textit{get-to-s}(t, c_0), \textit{pick-up}(t, c_0, p), \textit{get-to-d}(t, c_1), \textit{drop}(t, c_1, p)\\ &\textit{deliver}(p, c_1) \rightarrow \textit{get-to-s}(t, c_1), \textit{pick-up}(t, c_1, p), \textit{get-to-d}(t, c_1), \textit{drop}(t, c_1, p)\end{aligned}$$

Consider *deliver*(p, c_0), the task to deliver package p at location c_0. It can be decomposed using one of the first two methods. They differ in the location where the package is picked up, which is reflected in the first two tasks of the right-hand side of the respective methods.

Consider the second method: *deliver*(p, c_0) is decomposed into the following task sequence: first the truck must reach the source location where it shall pick up the package, which here is c_1, i.e., it must fulfill the task *get-to-s*(t, c_1). Then it picks up p at that location (fulfilling the primitive task *pick-up*(t, c_1, p)). Then it needs to reach the destination location where the package shall be dropped (*get-to-d*(t, c_0) and *drop*(t, c_0, p)).

We use different tasks/methods for reaching the source and destination locations to make the example more interesting later on. One could in principle use the same task/method set, of course. Both *get-to-s* and *get-to-d* enable recursion. Since there are only two locations in this simple example, introducing a recursive structure for moving is of course not necessary. However, given an instance with more locations, it might be necessary.

The task *get-to-s* can be fulfilled by first recursively calling *get-to-s*, followed by a primitive *drive* action:

$$\begin{aligned}
&\textit{get-to-s}(t, c_0) \rightarrow \textit{get-to-s}(t, c_1), \textit{drive}(t, c_1, c_0)\\
&\textit{get-to-s}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0)\\
&\textit{get-to-s}(t, c_1) \rightarrow \textit{get-to-s}(t, c_0), \textit{drive}(t, c_0, c_1)\\
&\textit{get-to-s}(t, c_1) \rightarrow \textit{drive}(t, c_0, c_1)
\end{aligned}$$

For *get-to-d*, it is the other way around, it can be fulfilled by first executing a primitive *drive* action, followed by a recursive call of *get-to-d*:

$$\begin{aligned}
&\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0)\\
&\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_0, c_1), \textit{get-to-d}(t, c_0)\\
&\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0), \textit{get-to-d}(t, c_0)\\
&\textit{get-to-d}(t, c_1) \rightarrow \textit{drive}(t, c_0, c_1)\\
&\textit{get-to-d}(t, c_1) \rightarrow \textit{drive}(t, c_0, c_1), \textit{get-to-d}(t, c_1)\\
&\textit{get-to-d}(t, c_1) \rightarrow \textit{drive}(t, c_1, c_0), \textit{get-to-d}(t, c_1)
\end{aligned}$$

Both recursive structures can be left by decomposing the respective *get-to-*$[s|d]$ task into a single *drive* action.

4. Translating TOHTN Problems to Classical Planning Problems

In this section, we investigate how to represent the language defined by the hierarchy (Solution Criteria 1 and 2) in a finite automaton. Since it is context-free, this might or might not be possible exactly. If we cannot automatically show that the represented language is regular, we build an automaton describing a superset of the original language. For now, we ignore the second language defined by the state transition semantics. We will come back to it later in Section 4.5.

4.1 Self-Embedding HTN Planning Problems

Given the language-based view on HTN planning from Section 2.3, we can now use properties and methods originally introduced for context-free grammars and apply them to HTN planning problems. First, we recap a sufficient criterion for a context-free grammar to describe a *regular* language. Since this is a semi-decidable problem, we cannot determine this exactly. The following definition is based on Nederhof (2000a, p. 19), but adapted in notation to HTN planning problems. It was originally introduced by Chomsky (1959). Let ε denote the empty string.

Definition 3 (Self-embedding grammars/problems)**.** *A grammar is* self-embedding *if there is some $c \in \mathcal{C}$ such that $c \rightsquigarrow \alpha c \beta$, for some $\alpha \neq \varepsilon$ and some $\beta \neq \varepsilon$. We call an HTN planning problem $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$* self-embedding *if and only if its grammar $G_h = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$ is self-embedding. When a grammar/problem is not self-embedding, we call it* non-self-embedding.

A grammar that is not self-embedding describes a regular language (Nederhof, 2000a, p. 19). Intuitively, this is because in every recursive cycle, it either creates symbols to the left *or* to the right, which both describe regular structures. Nederhof (2000a) introduces the following syntactical test that determines whether a grammar is self-embedding.

Definition 4 (Recursive symbols N^r)**.** *Given a grammar $G_h = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$, the set of recursive symbols is defined as $N^r = \{c \in \mathcal{C} \mid \exists \alpha, \beta : c \rightsquigarrow \alpha c \beta\}$. We divide these symbols into partitions $\overline{N} = \{N_1, N_2, \ldots, N_k\}$ such that $c_a, c_b \in N^r$ are in the same partition if and only if they can be decomposed into each other, i.e., $\exists \alpha_1, \beta_1, \alpha_2, \beta_2 : c_a \rightsquigarrow \alpha_1 c_b \beta_1 \wedge c_b \rightsquigarrow \alpha_2 c_a \beta_2$.*

The partition $\overline{N}$ divides the decomposition hierarchy into its strongly connected components (SCCs), i.e., tasks that can be decomposed into each other (using a sequence of decompositions) are in the same partition.

Running Example. The decomposition structure of our example is captured in the graph in Figure 3. It contains the tasks of the problem as nodes. Two nodes t_1 and t_2 are connected by an edge if and only if there is a method decomposing t_1 into a task sequence containing t_2. In HTN planning, this graph is commonly called *decomposition graph*. The partitions are given in the clusters N_0 to N_3. The recursive tasks and their clusters are the following:

$$
\begin{aligned}
N_0 &= \{\textit{get-to-s}(t, c_0), \textit{get-to-s}(t, c_1)\} \\
N_1 &= \{\textit{get-to-d}(t, c_1)\} \\
N_2 &= \{\textit{get-to-d}(t, c_0)\} \\
N_3 &= \{\textit{logistics-problem}()\}
\end{aligned}
$$

Next, Nederhof characterizes the recursion in partitions, which might be *left*, *right*, *cyclic*, or *self recursive*.

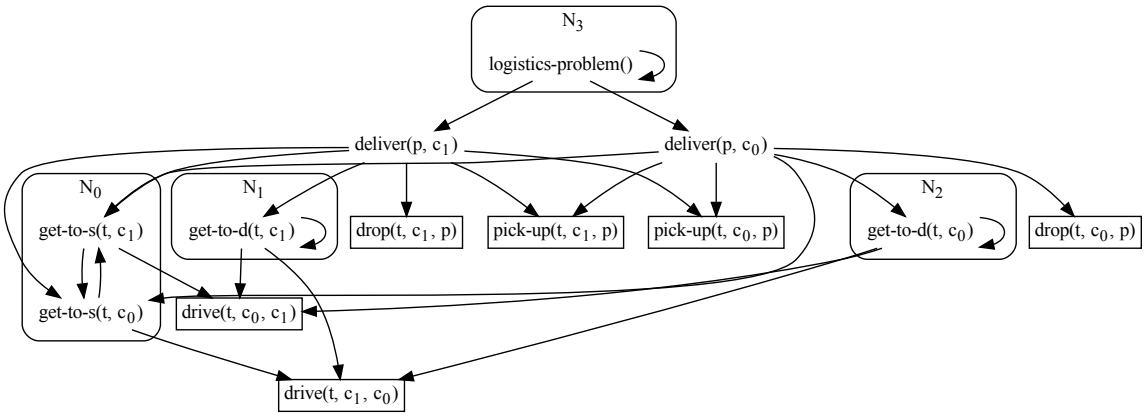

Figure 3: Decomposition graph of our running example. Actions are given in boxes, abstract tasks without. Clusters are depicted as rounded rectangles.

Definition 5 (Recursion of N_i)**.** *A partition $N_i \in \overline{N}$ is left generating, written $lg(N_i)$, if and only if $\exists(c_a, \alpha c_b \beta) \in \mathcal{M}$ such that $c_a, c_b \in N_i$ and $\alpha \neq \varepsilon$. It is right generating, $rg(N_i)$, if and only if $\exists(c_a, \alpha c_b \beta) \in \mathcal{M}$, $c_a, c_b \in N_i$ and $\beta \neq \varepsilon$. N_i is:*

- left recursive *if and only if $\neg lg(N_i)$ and $rg(N_i)$ hold,*
- right recursive *if and only if $lg(N_i)$ and $\neg rg(N_i)$ hold,*
- cyclic *if and only if $\neg lg(N_i)$ and $\neg rg(N_i)$ hold, and*
- self recursive *if and only if $lg(N_i)$ and $rg(N_i)$ hold.*

Running Example. The partitions in our example have the following recursive structures:

$$\begin{aligned} \textit{left recursive}: &\ N_0 \\ \textit{right recursive}: &\ N_1, N_2, N_3 \\ \textit{cyclic}: &\ - \\ \textit{self recursive}: &\ - \end{aligned}$$

A self recursive partition is a necessary and sufficient condition that the underlying grammar is self-embedding (Nederhof, 2000a, p. 20). The check further identifies those partitions that need to be modified to turn a self-embedding grammar into a *non*-self-embedding, i.e., into a grammar that describes a regular language.

4.2 Properties of HTN Benchmarks

Now that we have these properties at hand, let us analyse the commonly-used HTN models and see which properties they have. Table 1 summarizes the properties of the benchmark set of the track on totally ordered HTN planning of the 2020 International Planning Competition. For each domain, it shows how many instances fall in the different categories. The first column lists the non-recursive instances, followed by three columns listing the recursive but not self-embedding instances. The latter are split in those that are left recursive but not right recursive, right recursive but not left recursive, and both. Be aware that an instance can be left and right recursive but not self-embedding, as long as there is not a single partition that is both left and right recursive. The next column lists the self-embedding instances. Our system was not able to ground all instances. Since we analyse the properties on the ground models, we cannot give results for those instances. They are given in the last column. Further, we found that the grounding process has impact on the properties. E.g. in the *Depots* domain, where we found that the grounder's *h2* invariant analysis (or more precisely the tighter pruning resulting from this) turned several instances into non-recursive instances, which are self-embedding without it.

From the overall 892 instances, we were able to ground and thus analyse 861. Most interestingly, 78% of these instances are not self-embedding, which means that we can use the exact translation without the approximation step. About 18% are even non-recursive.

4.3 Superset Approximation of the Solution Set

When there are self-embedding partitions in a model, we modify these parts of the model to ensure that the resulting language is regular. We do this using an approximation introduced

		non-recursive	recursive				unknown
			non-self-emb.			self-emb.	
			left	right	l. & r.		
Assembly	30	-	-	30	-	-	-
Barman	20	20	-	-	-	-	-
BW-GTOHP	30	1	-	-	-	29	-
BW-HPDDL	30	-	-	30	-	-	-
Childsnack	30	26	-	-	-	-	4
Depots	30	20	-	-	-	10	-
Elevator	147	-	-	147	-	-	-
Entertainment	12	5	4	-	3	-	-
Factories	20	-	-	20	-	-	-
Freecell	60	-	-	-	-	60	-
Hiking	30	-	-	-	26	-	4
Logistics	80	-	-	80	-	-	-
Minecraft Pl	20	-	-	7	-	-	13
Minecraft Reg	59	49	-	-	-	-	10
Monroe-FO	20	-	-	-	-	20	-
Monroe-PO	20	-	-	-	-	20	-
Multiarm-BW	74	-	-	74	-	-	-
Robot	20	-	-	20	-	-	-
Rover	30	2	-	-	-	28	-
Satellite	20	-	-	-	-	20	-
Snake	20	-	-	20	-	-	-
Towers	20	-	-	20	-	-	-
Transport	40	-	40	-	-	-	-
Woodworking	30	30	-	-	-	-	-
	892	153	44	448	29	187	31

Table 1: Properties of the IPC 2020 benchmark set.

by Nederhof (2000b, p. 9), which increases the set of words of a language. The approximation is applied to each N_i identified as self recursive.

When we recall what caused a partition N_i to be self recursive, it was that it contains at least one rule that is left-generating and one that is right-generating. This enables an interplay of the rules resulting in languages like $\{a^n\ b\ a^n \mid n \geq 0\}$, which is context-free (and not regular). However, as we will see in our third example in this section, such a structure can also result in a regular language.

The approximation makes it possible to generate the parts (in the example from above, these are the a^n before the b and the a^n after the b) independent of each other, guaranteeing the resulting language to be regular. We next give the definition of the approximation by Nederhof (2000b, p. 9) and go through three examples afterwards.

Definition 6 (Approximation by Nederhof, 2000b)**.** *Let a-e be tasks part of N_i, all x_is are not part of N_i, α and β are sequences of tasks (that might or might not be part of N_i).*

1. *Add new non-terminals* $a_b^{\uparrow}$ $a_b^{\downarrow}$ $a_b^{\leftarrow}$ $a_b^{\rightarrow}$ *for all* $a, b \in N_i$
2. *Add the following methods for all* $a, b, c, d, e \in N_i$

$$(a, a_a^{\uparrow}) \tag{2.1}$$
$$(a_b^{\uparrow}, a_c^{\leftarrow} x_1 \dots x_m c_b^{\downarrow}), \forall (c, x_1 \dots x_m) \in M \tag{2.2}$$
$$(a_b^{\downarrow}, c_a^{\rightarrow} x_1 \dots x_m e_b^{\uparrow}), \forall (d, \alpha c x_1 \dots x_m e \beta) \in M \tag{2.3}$$
$$(a_b^{\downarrow}, b_a^{\rightarrow}) \tag{2.4}$$
$$(a_b^{\leftarrow}, x_1 \dots x_m c_b^{\leftarrow}), \forall (a, x_1 \dots x_m c \beta) \in M \tag{2.5}$$
$$(a_a^{\leftarrow}, \varepsilon) \tag{2.6}$$
$$(a_b^{\rightarrow}, c_b^{\rightarrow} x_1 \dots x_m), \forall (a, \alpha c x_1 \dots x_m) \in M \tag{2.7}$$
$$(a_a^{\rightarrow}, \varepsilon) \tag{2.8}$$

3. *Remove* (a, α) *from* M

To illustrate the effect of the approximation, we now go through three examples. We start by two context-sensitive languages to show the effect on the solution set. Then we have a look at our running example.

Example 1. Consider a grammar $G_1 = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$, where $\mathcal{C} = \{A\}$, $\mathcal{A} = \{a, b\}$, $\mathcal{M} = \{(A, b), (A, aAa)\}$, and $c_I = A$. It describes the given language $\{a^n\ b\ a^n \mid n \geq 0\}$, which is context-free. The approximation results in the following set of production rules. On the right it is given which part of the definition caused the respective rule.

$$A \to A_A^{\uparrow} \tag{2.1}$$
$$A_A^{\uparrow} \to A_A^{\leftarrow} b A_A^{\downarrow} \tag{2.2}$$
$$A_A^{\downarrow} \to A_A^{\rightarrow} \tag{2.4}$$
$$A_A^{\leftarrow} \to a A_A^{\leftarrow} \tag{2.5}$$
$$A_A^{\leftarrow} \to \varepsilon \tag{2.6}$$
$$A_A^{\rightarrow} \to A_A^{\rightarrow} a \tag{2.7}$$
$$A_A^{\rightarrow} \to \varepsilon \tag{2.8}$$

We can simplify these rules to the following set (where $X \to \alpha \mid \beta$ denotes that X can be decomposed into α or into β):

$$A \to A_A^{\leftarrow} b A_A^{\rightarrow}$$
$$A_A^{\leftarrow} \to a A_A^{\leftarrow} \mid \varepsilon$$
$$A_A^{\rightarrow} \to A_A^{\rightarrow} a \mid \varepsilon$$

Comparing the languages of the grammar before and after approximation, we see that the new rules enable the generation of the a's left of the b independently from those right of the b, describing the regular language $\{a^n\ b\ a^m \mid n, m \geq 0\}$.

$$
\begin{array}{llll}
S \to A \times B & A^{\downarrow}_{A} \to A^{\rightarrow}_{A} + B^{\uparrow}_{A} & A^{\leftarrow}_{A} \to (A^{\leftarrow}_{A} & A^{\rightarrow}_{A} \to B^{\rightarrow}_{A}) \\
A \to A^{\uparrow}_{A} & B^{\downarrow}_{A} \to A^{\rightarrow}_{B} + B^{\uparrow}_{A} & B^{\leftarrow}_{A} \to [A^{\leftarrow}_{A} & A^{\rightarrow}_{B} \to B^{\rightarrow}_{B}) \\
B \to B^{\uparrow}_{B} & A^{\downarrow}_{B} \to A^{\rightarrow}_{A} + B^{\uparrow}_{B} & A^{\leftarrow}_{A} \to \varepsilon & B^{\rightarrow}_{A} \to A^{\rightarrow}_{A}] \\
A^{\uparrow}_{A} \to A^{\leftarrow}_{A}\ a\ A^{\downarrow}_{A} & B^{\downarrow}_{B} \to A^{\rightarrow}_{B} + B^{\uparrow}_{B} & B^{\leftarrow}_{B} \to \varepsilon & B^{\rightarrow}_{B} \to A^{\rightarrow}_{B}] \\
B^{\uparrow}_{A} \to B^{\leftarrow}_{A}\ a\ A^{\downarrow}_{A} & A^{\downarrow}_{A} \to A^{\rightarrow}_{A} & & A^{\rightarrow}_{A} \to \varepsilon \\
B^{\uparrow}_{B} \to B^{\leftarrow}_{A}\ a\ A^{\downarrow}_{B} & B^{\downarrow}_{A} \to A^{\rightarrow}_{B} & & B^{\rightarrow}_{B} \to \varepsilon \\
B^{\uparrow}_{A} \to B^{\leftarrow}_{B}\ b\ B^{\downarrow}_{A} & A^{\downarrow}_{B} \to B^{\rightarrow}_{A} & & \\
B^{\uparrow}_{B} \to B^{\leftarrow}_{B}\ b\ B^{\downarrow}_{B} & B^{\downarrow}_{B} \to B^{\rightarrow}_{B} & &
\end{array}
$$

Figure 4: Rules resulting from the approximation as presented by Nederhof (2000b, p.11).

Example 2. Next we consider an example introduced by Nederhof (2000b, p. 9–10). Let the grammar $G_2 = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$ be a context-free grammar, where $\mathcal{C} = \{S, A, B\}$, $\mathcal{A} = \{a, b, (,), [,]\}$, $c_I = S$, and the production rules containing the following rules:

$$
\begin{aligned}
S &\to A \times B \\
A &\to (A + B) \\
A &\to a \\
B &\to [A] \\
B &\to b
\end{aligned}
$$

It describes a simple language of algebraic expressions. Examples for words that could be derived from it are:

- $a \times b$
- $(a + b) \times b$
- $a \times [(a + b)]$
- $(a + [(a + b)]) \times [(a + b)]$

The cluster that is self recursive includes the non-terminals A and B. When we apply the approximation, it results in the rules shown in Figure 4 (note that the rules have been simplified). Since we now have a regular language, we can visualize it by creating a finite automaton accepting it. The determinized and minimized automaton is given in Figure 5. As we can see, some structure of the original language is still included, e.g., that it is not possible to have two "a"s or "b"s next to each other, or a "$\times$" next to a "(". Further, all words that we derived from the original grammar are accepted, but many other words are also accepted that are not part of the original language, like "$(a \times b$". This example shows that the number of opening and closing parenthesis does not need to match anymore. While

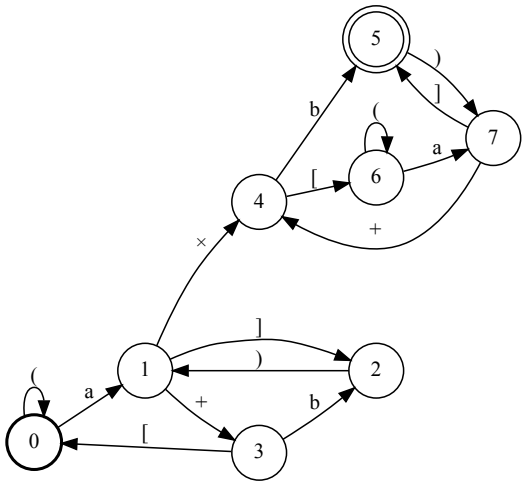

Figure 5: Determinized and minimized automaton accepting the language generated by the rules resulting from the approximation of Nederhof's example.

the language in this example is changed by the approximation, this is not necessarily the case. We will later on create a variation of our running example for which this is the case.

Figure 6 shows how the word "$(a+b)\times b$" can be derived from the original grammar (left) and from the transformed one (right). Consider the non-terminal symbols newly introduced by the approximation: the root of a subtree derived from the approximated rules is labeled with an arrow pointing upwards, e.g., $A^{\uparrow}_{A}$. It is then decomposed into one symbol with an arrow to the left, from which the left part is generated (e.g. $A^{\leftarrow}_{A}$) and one pointing down (e.g. $A^{\downarrow}_{A}$) from which the middle part as well as the right part of the original rule is derived. The middle part can e.g. be the "+" symbol in the $A \rightarrow (A + B)$ rule. While the left and right part were linked to each other in the original grammar such that there will be as many closing as opening parenthesis, these parts can now be generated separately. This leads to words like "$(a \times b$" from our example. The A in symbols like $B^{\uparrow}_{A}$ and $B^{\downarrow}_{A}$ indicates that the root of the subtree in the original grammar was labeled A, while the B is the last symbol treated (Nederhof, 2000b, p. 11).

Running Example. Now we come back to our running example. In the original example, the initial task was first decomposed into a sequence of one or more *delivery* tasks (see partition N_3 in Figure 3) by the following methods:

$$\begin{aligned}
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_0), \textit{logistics-problem}()\\
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_0)\\
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_1), \textit{logistics-problem}()\\
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_1)
\end{aligned}$$

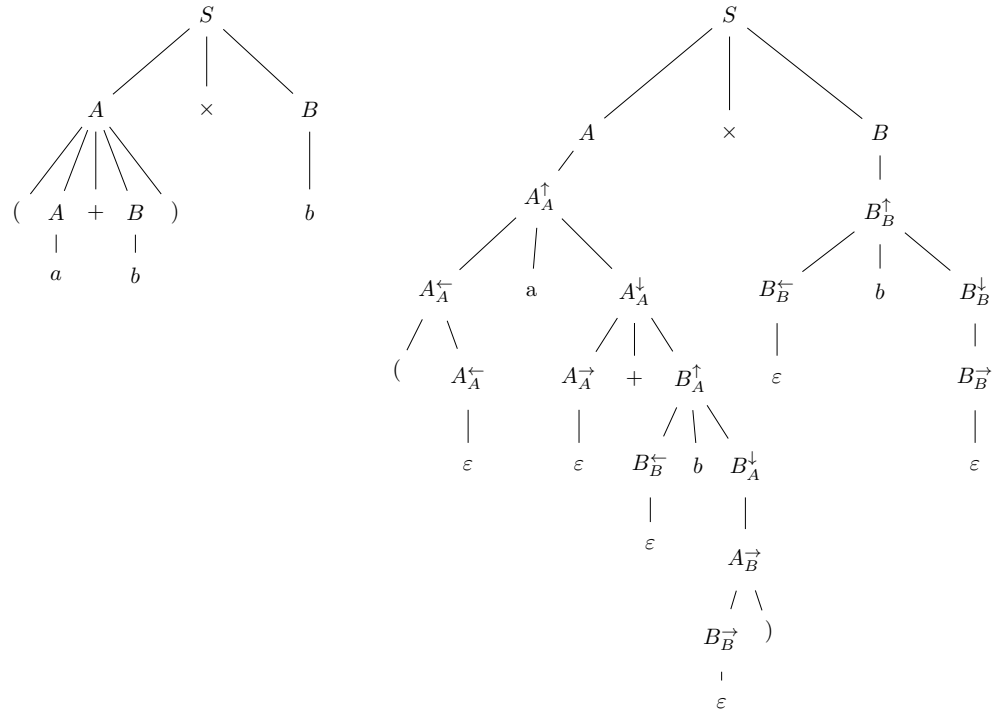

Figure 6: Example for a word derived from the original grammar (left) and from the transformed grammar (right).

For the sake of argument, let us replace these methods by the following ones:

$$\begin{aligned}
&\textit{logistics-problem}() \rightarrow \textit{logistics-problem}(), \textit{logistics-problem}()\\
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_0)\\
&\textit{logistics-problem}() \rightarrow \textit{deliver}(p, c_1)
\end{aligned}$$

The result in terms of the generated language is the same: we can generate a sequence of one or more *delivery* tasks (which is a regular language). However, the result of our analysis changes to the following:

$$\begin{aligned}
\textit{left recursive}: &\ N_0\\
\textit{right recursive}: &\ N_1, N_2\\
\textit{cyclic}: &\ -\\
\textit{self recursive}: &\ N_3
\end{aligned}$$

N_3 (which contains *logistics-problem*()) is now self recursive, the overall grammar is self-embedding, and we need to apply the approximation. When we do (and simplify the resulting rules), we get the following decomposition rules (we write *lp* instead of *logistics-problem*()

to improve readability; at the right you see which approximation rule produced the line):

$$lp \rightarrow lp^{\uparrow}_{lp} \tag{2.1}$$

$$lp^{\uparrow}_{lp} \rightarrow deliver(p, c_0), lp^{\downarrow}_{lp} \tag{2.2}$$

$$lp^{\uparrow}_{lp} \rightarrow deliver(p, c_1), lp^{\downarrow}_{lp} \tag{2.2}$$

$$lp^{\downarrow}_{lp} \rightarrow lp^{\uparrow}_{lp} \tag{2.3}$$

$$lp^{\downarrow}_{lp} \rightarrow \varepsilon \tag{2.4}$$

With the first rule, our original task is decomposed into $lp^{\uparrow}_{lp}$. This can then be decomposed into a *deliver* task followed by $lp^{\downarrow}_{lp}$ using one of the following two rules. $lp^{\downarrow}_{lp}$ can either be deleted using the last rule, or decomposed into $lp^{\uparrow}_{lp}$, forming a recursive cycle. In this example, the approximation does not change the language.

4.4 From Non-Self-Embedding HTN Problem to Finite Automaton

Now we have a non-self-embedding grammar, either the original grammar, or its approximation. Next we define a finite automaton $F = (Q, \Sigma, \Delta, q_0, q_F)$, where Q is a set of states, Σ a set of symbols (the alphabet), $\Delta : Q \times \Sigma \rightarrow 2^Q$ the transition function, q_0 the initial state, and $q_F \subseteq Q$ the set of final states. All sets are finite.

Let $F = (Q, \Sigma, \Delta, q_0, \{q_f\})$ be a finite automaton. Next we build a finite automaton that accepts the language of a non-self-embedding context-free grammar. We use the algorithm introduced by Nederhof (2000a), it is given in Algorithm 1 (adapted to our HTN notation).

As input, it gets two states q_0 and q_1 as well as a sequence of symbols α (which are in our case tasks). The algorithm constructs an automaton which captures the intermediate steps necessary to reach q_1 starting from q_0. We call the algorithm initially with q_0 and q_f. When the sequence α is empty, a new ε transition between the two states is added (line 2). When it is a terminal symbol a (in our case, an action), a transition is added that is labeled with a (line 3). When the algorithm is called with a sequence of more than one symbol, a novel state q is added (line 5), and the algorithm is called recursively. The intermediate state q can be reached from q_0 by parsing the first symbol of the sequence, and the second state q_1 is reached from q by the rest of the sequence.

The remainder of the algorithm handles abstract tasks. First consider the most simple case given at the bottom (line 24-26). It is concerned with tasks not contained in a partition N_i, i.e., tasks that are not recursive. Consider c_a is such a symbol, and the algorithm is called with the two states q_0, q_1, and c_a. The algorithm is called once for each method decomposing c_a, reflecting that one of the methods needs to be used to decompose the task. As a result, in the automaton, the sequence of tasks belonging to the decomposition of the tasks of one method needs to be parsed to reach the state q_1 when starting in q_0.

Consider the automaton given in Figure 7. It illustrates the automaton generated when calling the algorithm for the non-recursive abstract task $deliver(p, c_0)$. To improve readability, we kept the abstract *get-to* tasks abstract. When really calling the algorithm, they would directly be replaced by the respective sub-automaton.

Algorithm 1 Algorithm by Nederhof (2000a, Figure 2) to translate (non-self-embedding) context-free grammars to finite automata.

```
1  procedure make_fa(q_0, α, q_1)
2      if α = ε then Δ = Δ ∪ {(q_0, ε, q_1)}
3      else if α = a, a ∈ A then Δ = Δ ∪ {(q_0, a, q_1)}
4      else if α = xβ, x ∈ N, β ∈ N*, |β| > 0 then
5          q = fresh_state
6          make_fa(q_0, x, q)
7          make_fa(q, β, q_1)
8      else
9          c_a = α                                   /* α is abstr. task */
10         if ∃i : c_a ∈ N_i then
11             for c_b ∈ N_i do q_{c_b} = fresh_state
12             if recursive(N_i) = left then
13                 for (c_c, x_1 ... x_m) ∈ M s.t. c_c ∈ N_i ∧ x_1, ..., x_m ∉ N_i do
14                     make_fa(q_0, x_1 ... x_m, q_{c_c})
15                 for (c_c, c_d x_1 ... x_m) ∈ M s.t. c_c, c_d ∈ N_i ∧ x_1, ..., x_m ∉ N_i do
16                     make_fa(q_{c_d}, x_1 ... x_m, q_{c_c})
17                 Δ = Δ ∪ {(q_{c_a}, ε, q_1)}
18             else
19                 for (c_c, x_1 ... x_m) ∈ M s.t. c_c ∈ N_i ∧ x_1, ..., x_m ∉ N_i do
20                     make_fa(q_{c_c}, x_1 ... x_m, q_1)
21                 for (c_c, x_1 ... x_m c_d) ∈ M s.t. c_c, c_d ∈ N_i ∧ x_1, ..., x_m ∉ N_i do
22                     make_fa(q_{c_c}, x_1 ... x_m, q_{c_d})
23                 Δ = Δ ∪ {(q_0, ε, q_{c_a})}
24         else
25             for (c_a, β) ∈ M do
26                 make_fa(q_0, β, q_1)
```

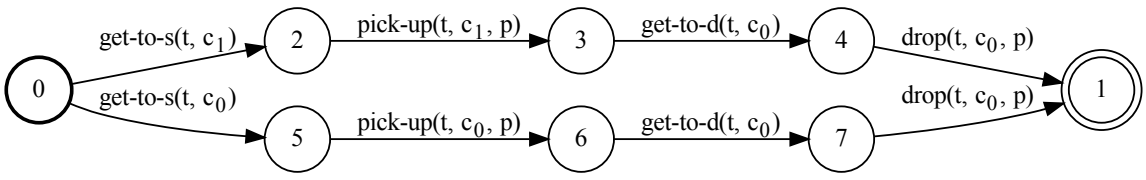

Figure 7: Sub-automaton for the task *deliver*(p, c_0).

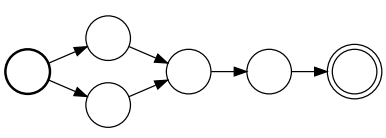

Figure 8: Optimized sub-automaton for *deliver*(p, c_0).

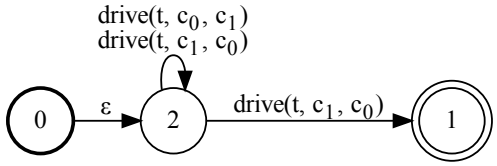

Figure 9: Automaton generated for the task *get-to-d*(t, c_0) belonging to partition N_2.

Let us have a closer look at the automaton. In our domain, there are the following two methods for the task, one for picking up the package at c_0 and one for doing so at c_1.

$$\textit{deliver}(p, c_0) \rightarrow \textit{get-to-s}(t, c_0), \textit{pick-up}(t, c_0, p), \textit{get-to-d}(t, c_0), \textit{drop}(t, c_0, p)$$
$$\textit{deliver}(p, c_0) \rightarrow \textit{get-to-s}(t, c_1), \textit{pick-up}(t, c_1, p), \textit{get-to-d}(t, c_0), \textit{drop}(t, c_0, p)$$

From the initial state 0, there is one path for each method. The last two tasks are the same for both methods, which leads to an automaton as given in Figure 8 after optimization.

After this example for a non-recursive abstract task, let us come back to Algorithm 1 and consider what happens for recursive tasks (starting in line 10). As a first step, one new state is created per symbol that belongs to the partition (line 11). We first consider the case of a partition N_i that is right-recursive (line 18-23). The lines 19-20 handle the case of methods that do not lead to a recursion, i.e., those leaving the partition. Such rules have the form $(c_c, x_1 \dots x_m)$, where c_c is a symbol from N_i and the symbols $x_1 \dots x_m$ are not contained in N_i. Starting from the state q_{c_c} introduced for c_c in line 11, one can leave (i.e., reach q_1) the recursive cycle by passing the automaton resulting from calling the algorithm with the sequence $x_1 \dots x_m$.

Now consider methods not leaving the partition N_i. Since it is a right recursive partition, these have the form $(c_c, x_1 \dots x_m c_d)$. For those, the state q_{c_d} introduced for c_d can be reached when starting from q_{c_c} by passing the automaton resulting from calling the algorithm with the sequence $x_1 \dots x_m$ (i.e., the final state of the sub-automaton cannot be reached this way). So far, the starting state q_0 has not been connected with the newly generated states. This is done via an ε transition in line 23 – the state corresponding to our α (set to c_a in line 9) can be reached from q_0.

Running Example. In our running example, consider partition N_2 containing the task *get-to-d*(t, c_0). The corresponding automaton is given in Figure 9. The algorithm is called with q_0 as initial state, $\alpha =$ *get-to-d*(t, c_0), and q_1 as final state. First, a new state for all tasks belonging to the partition is added (here, this is only one task, so only state 2 is added). For this task, there are the following three methods in the domain:

$$\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0)$$
$$\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_0, c_1), \textit{get-to-d}(t, c_0)$$
$$\textit{get-to-d}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0), \textit{get-to-d}(t, c_0)$$

The first one has the form *get-to-d*$(t, c_0) \rightarrow$ *drive*(t, c_1, c_0), it leaves the recursion when passing the automaton generated from the action sequence $\alpha =$ *drive*(t, c_1, c_0), i.e., it leads to an edge between the state introduced for *get-to-d*(t, c_0) (labeled 2 in Figure 9) and state 1.

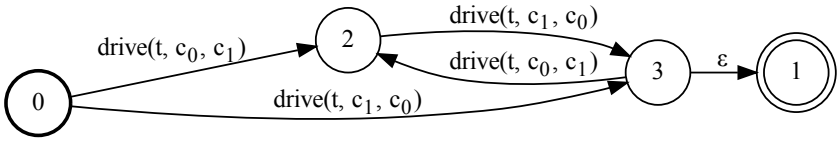

Figure 10: Automaton generated for the task *get-to-s*(t, c_0) belonging to partition N_2.

The two recursive methods have the form *get-to-d*$(t, c_0) \rightarrow$ *drive*(t, c_0, c_1), *get-to-d*(t, c_0), i.e., they connect the states representing the abstract tasks on the left and on the right side of the methods (both are *get-to-d*(t, c_0)) with the automaton generated by *drive*(t, c_0, c_1) and *drive*(t, c_1, c_0), respectively. This leads to the self-loops given at the top.

Now consider the case of left recursive partitions (line 12-17). It is symmetric to the case we just discussed. Non-recursive methods (line 13-14) do not leave the automaton, but enter it. For recursive methods, the sub-automaton needs to be passed *before* the one of the decomposed task (line 15-16), and the state q_{c_a} corresponding to the task in α is connected to the final state q_1 by an ε transition.

Running Example. In our example, only partition N_0 is left recursive. It contains the two tasks *get-to-s*(t, c_0) and *get-to-s*(t, c_1). Consider the algorithm is called with initial state q_0, $\alpha =$ *get-to-s*(t, c_0) (the first task from N_0), and the final state q_1. The corresponding automaton is given in Figure 10. First, a state for each task in the partition is generated (here, state 2 for *get-to-s*(t, c_1) and state 3 for *get-to-s*(t, c_0)). In the domain, there are the following methods belonging to the two tasks:

$$
\begin{aligned}
&\textit{get-to-s}(t, c_0) \rightarrow \textit{drive}(t, c_1, c_0)\\
&\textit{get-to-s}(t, c_0) \rightarrow \textit{get-to-s}(t, c_1), \textit{drive}(t, c_1, c_0)\\
&\textit{get-to-s}(t, c_1) \rightarrow \textit{drive}(t, c_0, c_1)\\
&\textit{get-to-s}(t, c_1) \rightarrow \textit{get-to-s}(t, c_0), \textit{drive}(t, c_0, c_1)
\end{aligned}
$$

For the first (non-recursive) method *get-to-s*$(t, c_0) \rightarrow$ *drive*(t, c_1, c_0), the state corresponding to the decomposed task *get-to-s*(t, c_0) (labelled 3) can be reached from the initial state (labeled 0) by the automaton generated from the right-hand side of the method (here only the single action *drive*(t, c_1, c_0)). The third method *get-to-s*$(t, c_1) \rightarrow$ *drive*(t, c_0, c_1), which is also non-recursive, leads to the transition connecting the initial state 0 with the state 2, which corresponds to the abstract task *get-to-s*(t, c_1).

The (recursive) method *get-to-s*$(t, c_0) \rightarrow$ *get-to-s*(t, c_1)*drive*(t, c_1, c_0) connects the state corresponding to the contained abstract tasks *get-to-s*(t, c_0) and *get-to-s*(t, c_1) by the automaton generated from the remaining parts of the right-hand side. This is state 2 (introduced for *get-to-s*(t, c_1)) and 3 (introduced for *get-to-s*(t, c_0)) by the transition labeled *drive*(t, c_1, c_0) (be aware that the left-hand side of the method forms the target of the transition). The transition introduced for the last method connects 3 to 2 and is labeled *drive*(t, c_0, c_1). Since we showed the automaton introduced for *get-to-s*(t, c_0), the corresponding state 3 is connected to the final state 1 by an ε transition. In the automaton generated for *get-to-s*(t, c_1), this would be 2.

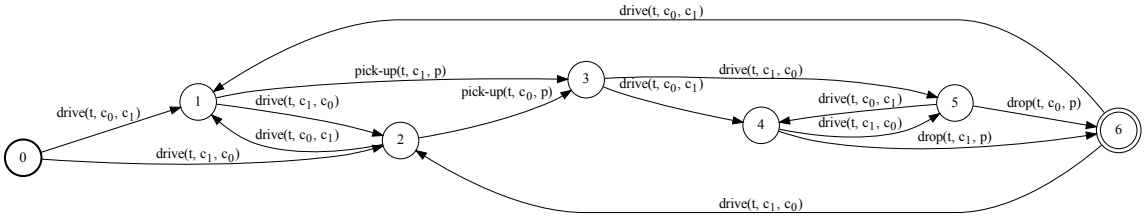

Figure 11: Automaton generated for the overall HTN planning problem, i.e., for the task *logistics-problem*().

Figure 11 shows the final automaton (without ε transitions, determinized and minimized) for the overall problem. Notice that transitions are labeled with the actions from the planning problem ($\Sigma \subseteq \mathcal{A}$).

When no approximation was necessary for a given problem, the automaton accepts the language L_h (defined by the hierarchy of the HTN problem). When approximation was applied, it accepts a superset of L_h. Further, it does not include constraints introduced by the state transition system of the problem, we need to intersect its language with the language L_c. This can e.g. be seen in the initial state of the automaton (0). While t is initially located at position c_0, the automaton also accepts solutions starting with action $drive(t, c_1, c_0)$. This is not caused by approximation (we did not approximate in this example), but by the fact that it accepts L_h and needs to be restricted to the intersection of L_h and L_c.

When we have a closer look at the automaton, we see that words/plans start with a sequence of one or more *drive* actions (states 0 to 2), followed by a *pick-up* action to reach state 3. The states 3 to 5 – again – accept a sequence of one or more *drive* actions. The final state 6 can then be reached by a *drop* action. When more than one package needs to be delivered, 6 can be left by a drive action. As can be seen, L_h accepts solutions where one or more packages are delivered. When there *is* more than one in the problem, L_c needs to ensure that all packages are delivered (e.g. by the goal condition like in this example).

4.5 Encoding as Classical Planning Problem

Now we have (1) the action definitions of the original problem and (2) a finite automaton capturing the (approximated or exact) hierarchy. Next, we create a classical planning problem where an action sequence is contained in the solution set if and only if

- it is accepted by the automaton, and
- it is applicable in the initial state and its application results in a goal state (using the transition semantics of the original actions from the HTN model).

Based on these properties, we show in Section 4.6 that our approach is complete, and with an additional verification step also sound. But first we explain our construction. The basic idea is straightforward: we maintain two separate parts of the state. One is the state of the original HTN planning problem. For an action to be applicable, its original preconditions need to hold. When it is applied, state transition regarding this part of the state is the same

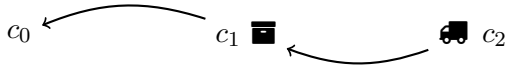

Figure 12: Illustration of the initial state of Example 3.

as in the original problem. When there is a state-based goal definition in the HTN planning problem, we add the respective condition to our definition here as well.

The second part of the state maintains the current state of the automaton. From its state features, exactly one holds at a time. An action a is applicable if and only if there is an (outgoing) transition from the respective state of the automaton. The action's effect mimics the transition in the automaton. Our new goal condition contains the goal state of the automaton[4].

Let $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$ be the HTN planning problem that we want to solve and G_h the grammar describing all action sequences reachable in the model via decomposition as introduced in Definition 2. Let $F = (Q, \mathcal{A}, \Delta, q_0, \{q_f\})$ be the automaton capturing the grammar G_h, and $\Pi_c{}' = (\mathcal{V}', \mathcal{A}', s_0{}', s_\star{}')$ the resulting classical problem that we encode. We define its elements as follows:

$$\begin{aligned}
&\mathcal{V}' = \mathcal{V} \cup \{v_Q\}, \text{ with } \mathcal{D}_{v_Q} = \{q \mid q \in Q\},\\
&s'_0 = s_0 \cup \{v_Q = q_0\},\\
&s'_\star = s_\star \cup \{v_Q = q_f\}
\end{aligned}$$

We add a single state variable v_Q for which the automaton's states form the domain. In the new initial state, v_Q is set to q_0, the initial state of the automaton. In the new goal definition, v_Q must equal q_f, the goal state of the automaton.

The action set is then defined as follows:

$$\mathcal{A}' = \{a_r^q \mid (q, a, r) \in \Delta\}$$

For each transition $(q, a, r) \in \Delta$ of the automaton, i.e., starting in state q, leading to state r, and labeled with a, we introduce an action a_r^q.

The preconditions and effects in the new problem are copies from the original HTN planning problem, but extended by $v_Q = q$ and $v_Q = r$, respectively:

$$\begin{aligned}
\forall a_r^q \in \mathcal{A}' : \ & \mathit{prec}'(a) = \mathit{prec}(a) \cup \{v_Q = q\}\\
& \mathit{eff}'(a) = \mathit{eff}(a) \cup \{v_Q = r\}
\end{aligned}$$

Our encoding is similar to one introduced by Chrpa and Barták (2016), which we will discuss in the related work section (Section 6).

4. The construction given above leads to exactly one goal state. In the implementation, minimization and determinization may lead to more than one goal state. We compile this away by adding dummy actions from the various goal states to a single new one.

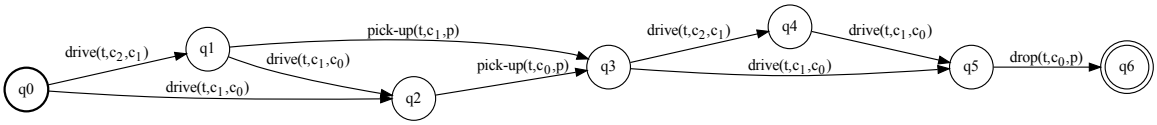

Figure 13: Automaton from Example 3.

Example 3. Let us have a look at an example. Like our running example it is a transport problem. However, we further simplified it to show the full classical encoding. The variables $\mathcal{V}$, actions $\mathcal{A}$, initial state s_0, and goal definition $s_\star$ are define as follows. The initial state is illustrated in Figure 12. The package shall be delivered at position c_0, but other than in the running example we encode this information into the decomposition hierarchy instead of the state-based goal. The remaining parts of the HTN model are not needed for the translation.

$$\begin{aligned}
\mathcal{V} =& \{l(p), l(t)\}, \text{ with } \mathcal{D}_{l(p)} = \{at(p, c_0), at(p, c_1), in(p, t)\}, \text{ and} \\
& \mathcal{D}_{l(t)} = \{at(t, c_0), at(t, c_1), at(t, c_2)\} \\
\mathcal{A} =& \{drive(t, c_1, c_0), drive(t, c_2, c_1), drop(t, c_0, p), pick\text{-}up(t, c_0, p), pick\text{-}up(t, c_1, p)\} \\
s_0 =& \{l(p) = at(p, c_1), l(t) = at(t, c_2)\} \\
s_\star =& \{\}
\end{aligned}$$

The actions come with the following preconditions and effects:

$$\begin{aligned}
prec(drive(t, c_1, c_0)) &\mapsto \{l(t) = at(t, c_1)\} \\
eff(drive(t, c_1, c_0)) &\mapsto \{l(t) = at(t, c_0)\} \\
prec(drive(t, c_2, c_1)) &\mapsto \{l(t) = at(t, c_2)\} \\
eff(drive(t, c_2, c_1)) &\mapsto \{l(t) = at(t, c_1)\} \\
prec(pick\text{-}up(t, c_0, p)) &\mapsto \{l(p) = at(p, c_0), l(t) = at(t, c_0)\} \\
eff(pick\text{-}up(t, c_0, p)) &\mapsto \{l(p) = in(p, t)\} \\
prec(pick\text{-}up(t, c_1, p)) &\mapsto \{l(p) = at(p, c_1), l(t) = at(t, c_1)\} \\
eff(pick\text{-}up(t, c_1, p)) &\mapsto \{l(p) = in(p, t)\} \\
prec(drop(t, c_0, p)) &\mapsto \{l(t) = at(t, c_0), l(p) = in(p, t)\} \\
eff(drop(t, c_0, p)) &\mapsto \{l(p) = at(p, c_0)\}
\end{aligned}$$

The automaton resulting from the transformation is given in Figure 13. The classical planning problem $\Pi_c{}' = (\mathcal{V}', \mathcal{A}', s_0', s_\star')$ as introduced in this section is defined as follows:

$$\begin{aligned}
\mathcal{V}' &= \{l(p), l(t)\} \cup \{v_Q\}, \text{ with } \mathcal{D}_{v_Q} = \{q0, \ldots, q6\} \\
\mathcal{A}' &= \{drive(t, c_2, c_1)^{q0}_{q1}, drive(t, c_1, c_0)^{q0}_{q2}, drive(t, c_1, c_0)^{q1}_{q2}, \\
&\quad pick\text{-}up(t, c_1, p)^{q1}_{q3}, pick\text{-}up(t, c_0, p)^{q2}_{q3}, drive(t, c_2, c_1)^{q3}_{q4}, \\
&\quad drive(t, c_1, c_0)^{q3}_{q5}, drive(t, c_1, c_0)^{q4}_{q5}, drop(t, c_0, p)^{q5}_{q6}\} \\
s_0' &= \{l(p) = at(p, c_1), l(t) = at(t, c_2)\} \cup \{v_Q = q0\} \\
s_\star' &= \{\} \cup \{v_Q = q6\}
\end{aligned}$$

The resulting model has as many actions as there are transitions in the automaton (9 in this case). Each action has the preconditions and effects as the original one from the HTN model. An additional precondition specifies in which state of the automaton it is applicable, an additional effect sets the automaton state resulting from the application.

$$
\begin{aligned}
prec'\left(drive(t,c_2,c_1)^{q0}_{q1}\right) &\mapsto prec\left(drive(t,c_2,c_1)\right) \cup \{v_Q = q0\}\\
eff'\left(drive(t,c_2,c_1)^{q0}_{q1}\right) &\mapsto eff\left(drive(t,c_2,c_1)\right) \cup \{v_Q = q1\}\\
prec'\left(drive(t,c_1,c_0)^{q0}_{q2}\right) &\mapsto prec\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q0\}\\
eff'\left(drive(t,c_1,c_0)^{q0}_{q2}\right) &\mapsto eff\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q2\}\\
prec'\left(drive(t,c_1,c_0)^{q1}_{q2}\right) &\mapsto prec\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q1\}\\
eff'\left(drive(t,c_1,c_0)^{q1}_{q2}\right) &\mapsto eff\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q2\}\\
prec'\left(pick\text{-}up(t,c_1,p)^{q1}_{q3}\right) &\mapsto prec\left(pick\text{-}up(t,c_1,p)\right) \cup \{v_Q = q1\}\\
eff'\left(pick\text{-}up(t,c_1,p)^{q1}_{q3}\right) &\mapsto eff\left(pick\text{-}up(t,c_1,p)\right) \cup \{v_Q = q3\}\\
prec'\left(pick\text{-}up(t,c_0,p)^{q2}_{q3}\right) &\mapsto prec\left(pick\text{-}up(t,c_0,p)\right) \cup \{v_Q = q2\}\\
eff'\left(pick\text{-}up(t,c_0,p)^{q2}_{q3}\right) &\mapsto eff\left(pick\text{-}up(t,c_0,p)\right) \cup \{v_Q = q3\}\\
prec'\left(drive(t,c_2,c_1)^{q3}_{q4}\right) &\mapsto prec\left(drive(t,c_2,c_1)\right) \cup \{v_Q = q3\}\\
eff'\left(drive(t,c_2,c_1)^{q3}_{q4}\right) &\mapsto eff\left(drive(t,c_2,c_1)\right) \cup \{v_Q = q4\}\\
prec'\left(drive(t,c_1,c_0)^{q3}_{q5}\right) &\mapsto prec\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q3\}\\
eff'\left(drive(t,c_1,c_0)^{q3}_{q5}\right) &\mapsto eff\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q5\}\\
prec'\left(drive(t,c_1,c_0)^{q4}_{q5}\right) &\mapsto prec\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q4\}\\
eff'\left(drive(t,c_1,c_0)^{q4}_{q5}\right) &\mapsto eff\left(drive(t,c_1,c_0)\right) \cup \{v_Q = q5\}\\
prec'\left(drop(t,c_0,p)^{q5}_{q6}\right) &\mapsto prec\left(drop(t,c_0,p)\right) \cup \{v_Q = q5\}\\
eff'\left(drop(t,c_0,p)^{q5}_{q6}\right) &\mapsto eff\left(drop(t,c_0,p)\right) \cup \{v_Q = q6\}
\end{aligned}
$$

Earlier in Section 2.2 on page 616 we mentioned that the state-based goal definition of HTN planning problems is usually empty (like in the given example). When adaption e.g. heuristics from classical planning to apply them in HTN planning, this is a problem, because those naturally depend on the goal definition. However, consider what happens in our approach. The goal of the problem is then to find a path of applicable actions through the automaton that reaches the automaton's goal state. The automaton captures the action sequences that may result from the decomposition process, the original action preconditions/effects maintain applicability. So this is exactly what we want to have in this case, an executable action sequence resulting from the decomposition process.

Now we investigate the theoretical properties of our classical FDR encoding. We need these results later on to show the properties of the overall TOAD system.

Theorem 2 (Soundness FDR Acceptance)**.** *Let* $\pi = a1_{r1}^{q1}, a2_{r2}^{q2}, \ldots, an_{rn}^{qn}$ *be a solution for* ${\Pi_c}'$*. Then, the action sequence* $a1, a2, \ldots, an$ *is accepted by* F*.*

Proof. We introduced a novel variable v_Q with $\mathcal{D}_{v_Q} = \{q \mid q \in Q\}$, which tracks the current state of the automaton. Since this variable has been newly introduced, only our also newly introduced effects change it. In each state of the planning problem, we are in exactly one state of the FA. In s_0' it is assigned to q_0, i.e., the initial state of the automaton. Actions in ${\Pi_c}'$ have the form a_r^q. An action a_r^q is in $\mathcal{A}'$ if and only if $(q, a, r) \in \Delta$. Its newly introduced precondition makes it applicable only if $v_Q = q$ holds, and its new effect assigns v_Q to r. Since $a1_{r1}^{q1}$ is applicable in s_0', it must be that $q1 = q_0$. And by construction, $(q1, a1, r1)$ must be in Δ. When $a2_{r2}^{q2}$ is applicable afterwards, it must be that $r1 = q2$. Further, $(q2, a2, r2)$ must be in Δ and so on. Since we made $v_Q = q_f$ part of the goal of ${\Pi_c}'$, we know that $qn' = q_f$ and that the automaton accepts the sequence. □

Theorem 3 (Soundness FDR Goal Completion for Π_h)**.** *Let* $\pi = a1_{r1}^{q1}, a2_{r2}^{q2}, \ldots, an_{rn}^{qn}$ *be a solution for* ${\Pi_c}'$*. Then the action sequence* $a1, a2, \ldots, an$ *is applicable and leads to a goal state in* Π_h*.*

Proof. Starting from the original HTN planning problem, we added a new part to the state definition consisting of exactly one variable, v_Q. New effects do not change other parts of the state.

It holds that $(s_0' \setminus \{(v_Q = q_0)\}) = s_0$. Regarding the original state features, preconditions did not change, i.e., for each action $\left(prec'(ai_{ri}^{qi}) \setminus \{(v_Q = qi)\}\right) = prec(ai)$. When $a1_{r1}^{q1}$ is applicable in s_0', then $a1$ must be in s_0.

Novel effects also only change v_Q, i.e., $\left(\mathit{eff}'(ai_{ri}^{qi}) \setminus \{(v_Q = ri)\}\right) = \mathit{eff}(ai)$. Therefore, when ignoring v_Q, the intermediate states in the action sequence are the same in ${\Pi_c}'$ and Π_h. The same holds for the goal condition. □

Theorem 4 (Completeness FDR)**.** *Let* $\pi = a1, a2, \ldots, an$ *be an action sequence that is*

1. *accepted by the automaton* F *and*
2. *that is applicable and leads to a goal state in* Π_h*.*

Then, there is a solution $\pi' = a1_{q2}^{q1}, a2_{q3}^{q2}, \ldots, an_{q(n+1)}^{qn}$ *in* ${\Pi_c}'$*.*

Proof. A state definition in ${\Pi_c}'$ consists of two parts, the original state from the HTN problem and the variable v_Q representing the current state in the automaton. The part from the original problem is unchanged, both the state variables and all preconditions and effects are the same. The newly added part only relates to v_Q. Therefore, if π is applicable and results in a goal state in Π_h (prerequisite 2), the original preconditions hold in π', and the original state-based goal is fulfilled afterwards.

Further we need to show that the action sequence fulfills the newly added part of the state/action definition. We know that it is accepted by F (prerequisite 1), therefore there

must be a state sequence $q1, q2, \ldots, qn$ such that $(qi, a_i, q(i+1)) \in \Delta$, and $q(n+1)$ is a goal state in F. By definition, $\mathcal{A}' = \{a_r^q \mid (q, a, r) \in \Delta\}$, i.e., there is an action a_r^q if and only if $(q, a, r) \in \Delta$. Therefore there must be a sequence $\pi' = a1_{q2}^{q1}, a2_{q3}^{q2}, \ldots, an_{q(n+1)}^{qn}$, which is applicable and which leads to a goal state with respect to v_Q. □

4.6 The Toad Planning System

Now we put the pieces together to an overall system, and show its properties afterwards. An overview is given in Figure 1 on page 614.

Definition 7 (Toad Planning System)**.** *Let Π_h be an HTN planning problem. We denote the following steps the* Toad translation*, which results in a classical planning problem $\Pi_c = (\mathcal{V}, \mathcal{A}, s_0, s_\star)$:*

1. *(If the problem Π_h is self-embedding,) the approximation of its grammar G_h as given in Section 4.3, followed by*
2. *the translation into an automaton as given in Algorithm 1, followed by*
3. *the translation to a planning problem as given in Section 4.5.*

We generate solutions for the resulting model using a classical planning system until one is identified as solution for the HTN problem by an HTN plan verifier.

Having the definition of the Toad system at hand, we are interested in its theoretical properties. Let Π_h be an HTN planning problem and Π_T the classical planning problem resulting from the Toad translation. For the following theorems, we make two assumptions:

Assumption 1. *To find plans in the* Toad *system, we use a classical planning system that eventually returns every solution to a given problem.*

More precisely, when we generate a first solution to the classical problem, verify it, and get the result that it is not a solution to the HTN planning problem, we assume to be able to get the next solution from the classical planning system, and will eventually get every single solution to the problem. This assumption assures that we can in the following proofs reason about the sets of solutions of (1) the original HTN problem and (2) the encoded classical one instead of incorporating a single planning system and how it generates solutions. Whether or not this property is fulfilled in practice is discussed in Section 7.2.

Assumption 2. *We assume that a sound and complete verification system is applied, i.e., a system that decides whether a given sequence of actions is a valid solution for an HTN planning problem.*

This assumption is no restriction, there are several systems in the literature that can be used (see Section 7.3). For totally ordered HTN planning as covered in this article, the task is solvable in polynomial time and resembles parsing a context-free language, for partially ordered HTN planning, the task is NP-complete (Behnke et al., 2015).

Recall that L_Π is the set of solutions of a planning problem Π, i.e., the action sequences that fulfilling all solution criteria (page 617). The following theorem states that all solutions to the original HTN problem are included in the solution set to the problem resulting from the Toad translation.

Theorem 5 (Completeness Toad)**.** *It holds that* $L_{\Pi_h} \subseteq L_{\Pi_T}$.

Proof. Let us go through the steps of the Toad translation, the application of the classical planning system and the HTN verifier.

- In the first step, G_h is re-written if it is self-embedding. However, we know that this is done in a way increasing the set of solutions (Nederhof, 2000a, 2000b), i.e., we have no problem for the completeness of the overall approach.
- The second step encodes it as a FA, for which we know that it accepts the language of the grammar (Nederhof, 2000a, 2000b).
- Next we encode it as a classical planning problem. From Theorem 4 we know that for every task sequence that (1) is accepted by the FA and (2) is applicable and leads to a goal state in the original HTN planning problem Π_h, there is a corresponding solution of the classical planning problem. By Assumption 1 we know that all these plans will eventually be returned by the planning system.
- By Assumption 2 we know that the verification system will not reject a valid solution to the HTN problem. □

Theorem 6 (Soundness Toad)**.** *Solutions returned by the* Toad *planning system are solutions for the HTN problem.*

Proof. Since solutions are verified, we know by Assumption 2 that the theorem holds. □

Besides this trivial case, there is a more interesting one, namely whether solutions returned from a translation without approximation are sound without verification.

Theorem 7 (Soundness Toad (exact translation))**.** *Solutions returned by the* Toad *planning system are sound without verification when no approximation is applied.*

Proof. Let us go through the steps of the overall process.

- When no approximation is used in Step 1 of the translation, we know that the automaton F generated in Step 2 exactly accepts the action sequences that might result from the decomposition.
- From Theorem 2, we know that for any solution for the Toad translation, the corresponding action sequence is accepted by F, i.e., it is a sequence that can be derived via decomposing the initial task. This means that the sequence fulfills the Solution Criteria 1 and 2 of HTN planning.
- From Theorem 3, we know that for any solution for the Toad translation, the corresponding action sequence is applicable and leads to a goal state in the HTN planning problem, which makes it fulfill Solution Criterion 3.

Thus, an action sequence returned by the classical planning system fulfills all three HTN solution criteria from Section 2.2, page 616. □

As a result, we may skip the verification step when no approximation was applied.

5. Relation of Self-Embedding and Tail-Recursiveness

With self-embedding, we introduced a novel criterion to check whether a given HTN planning problem can be translated into a classical planning problem exactly (i.e., without approximation or bounding the problem). In the HTN literature, there is already such a criterion, namely *tail-recursiveness* (Alford et al., 2012).

Next we investigate the relation of these criteria. Alford et al. (2012) introduced a syntactic test called $\leq_r$-stratifiability to check whether an HTN planning problem is tail-recursive or not. The following definition is taken from Alford et al. (2016, p. 5) and adapted to totally ordered HTN planning.

Definition 8 ($\leq_r$-stratifiability by Alford et al., 2016)**.** *A totally ordered HTN planning problem is $\leq_r$-stratifiable if and only if there exists a total preorder $\leq_r$ on the tasks $\mathcal{N}$ such that for all methods $(c, t_1 t_2 \dots t_n) \in \mathcal{M}$ with $|t_1 t_2 \dots t_n| > 0$, the following holds:*

1. $t_n \leq_r c$
2. $t_i <_r c$ *for* $1 \leq i < n$

Such methods can only contain full recursion on their last task. From all other tasks contained in the method, the decomposed task c is not reachable anymore.

Theorem 8. *Every totally ordered tail-recursive HTN problem is non-self-embedding.*

Proof. We need to show that, given a grammar $G_h = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$ of a totally ordered tail-recursive HTN planning problem $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$, there is no partition N_i of $\overline{N}$ that is both $lg(N_i)$ and $rg(N_i)$. Then, the partition is not self recursive and thus the problem not self-embedding.

Assume there is a partition N_i that is right generating, i.e., $rg(N_i)$ holds.

1. According to Definition 5, this implies that there must be a grammar rule $(c_a, \alpha c_b \beta) \in \mathcal{M}$ such that $c_a, c_b \in N_i$ and $\beta \neq \varepsilon$. This means that we know that c_b is not the last task.
2. Since we know that c_b is a non-last task, we know from Definition 8 that $c_a <_r c_b$, where $<_r$ is a preorder.
3. According to Definition 4, two symbols are in the same partition if and only if there exist $\alpha_1, \beta_1, \alpha_2, \beta_2$ such that $c_a \rightsquigarrow \alpha_1 c_b \beta_1$ and $c_b \rightsquigarrow \alpha_2 c_a \beta_2$. This means that we know that, starting from c_b, we can get back to c_a.
4. This means that there is a sequence of methods $c_a \rightsquigarrow \alpha_1 c_b \beta_1 \rightsquigarrow \alpha_2 c_c \beta_2 \rightsquigarrow \dots \rightsquigarrow \alpha_n c_a \beta_n$. Since $c_a <_r c_b$ and due to the transitivity of a preorder relation, we know that $c_a <_r c_a$, which is a contradiction to the reflexivity of preorders.

Our assumption that a totally ordered tail-recursive HTN problem can be right generating must be wrong. Thus, such a problem cannot be self-embedding either. □

Theorem 9. *Let $\Pi_h = (\mathcal{V}, \mathcal{C}, \mathcal{A}, \mathcal{M}, c_I, s_0, s_\star)$ be a totally ordered HTN problem and $G_h = (\mathcal{C}, \mathcal{A}, \mathcal{M}, c_I)$ its grammar. When there is no partition N_i in the grammar's recursive structure that is right generating (i.e., $\neg \exists N_i \in \overline{N} : rg(N_i)$), then Π_h is tail-recursive.*

Proof. We first define an ordering relation on the tasks and then show that it is a total preorder fulfilling the two conditions from Definition 8.

1. We construct the graph $G_{dec} = (\mathcal{N}, E)$ with $E = \{(a, b) \mid a, b \in \mathcal{N}$ and $\exists(a, \omega) \in \mathcal{M}$ with $b \in \omega\}$.

   G_{dec} captures the decomposition structure of Π_h. The task names form the nodes of the graph, while two nodes a and b are connected if and only if there is a method decomposing the task a from the problem into a task sequence containing b.

2. First we compute the strongly connected components (SCCs) of G_{dec}, and then its condensation , i.e., we contract each SCC to a single vertex. The result is a directed, acyclic graph that we call G_{scc}.

   Be aware the connection of G_{scc} and $\overline{N}$: each partition N_i out of $\overline{N}$ forms an scc, while only recursive tasks are contained in some N_i, i.e., the condensed nodes form a superset of $\overline{N}$.

3. We compute a topological ordering of the sccs, let $\overline{scc} = (scc_0, scc_1, \dots scc_k)$ be the resulting sequence, where the initial task is contained in scc_0.

4. We define the following ordering relation $\preceq$:

   - $a \preceq b$ holds if and only if the tasks a and b are in the same scc (i.e., as a special case, it holds that $a \preceq a$).
   - Between two tasks a and b from different sccs scc_i (with $a \in scc_i$) and scc_j (with $b \in scc_j$), it holds that $a \prec b$ if and only if $i < j$ in $\overline{scc}$.

The ordering relation defined in 4 is a total preorder on the tasks:

- All tasks are contained in our graph. We defined ordering relations between tasks from the same and those from different sccs. Our ordering is total.
- By definition, it is reflexive, i.e., $a \preceq a$ holds for all tasks.
- All tasks from the same scc are equal, while ordering relations between tasks from different sccs stem from a total order of the sccs. Thus the ordering relation is also transitive.

What is left to show is that the ordering relation fulfills the two conditions of Definition 8, i.e., that for all methods $(c, t_1 t_2 \dots t_n) \in \mathcal{M}$ with $|t_1 t_2 \dots t_n| > 0$, the following holds:

(c1) $t_n \leq_r c$

(c2) $t_i <_r c$ for $1 \leq i < n$

Let us first recap what we know from the prerequisites:

1. A partition N_i is right generating if and only if $\exists(c_a, \alpha c_b \beta) \in \mathcal{M}$, $c_a, c_b \in N_i$ and $\beta \neq \varepsilon$. We know that there is no such method, i.e., $\forall m \in \mathcal{M}$ with $c_a, c_b \in N_i$ and $m = (c_a, \alpha c_b \beta)$, it holds that $\beta = \varepsilon$.

2. There exist two kinds of methods:
   (a) recursive methods of the form $m = (c_a, \alpha c_b)$ with $c_a, c_b \in N_i$, and
   (b) methods not leading to a recursion of the form (c_a, α) with $\forall n \in \alpha : n \notin N_i$.

By Definition 5, two symbols are in the same partition if and only if they can be decomposed into each other. In the first case (2a), we know that c_a and c_b are in the same partition N_i, thus they can be decomposed into each other, i.e., they are in the same scc in G_{dec}, leading to an ordering relation $c_a \leq c_b$ between them. Since c_b must be the last task, this is in line with condition (c1).

For the other symbols $c_c \in \alpha$, we know that they are not in the same partition as c_a, thus they cannot be decomposed back into c_a. Since c_a can be decomposed into c_c but not vice versa, this means that – in our ordering – c_a must be left of c_c, i.e., it holds that $c_a < c_c$. The same holds for case (2b).

Thus, we have defined a total preorder on the tasks that fulfills the conditions (c1) and (c2), i.e., we have shown that totally ordered HTN problems that have no right generating partitions are tail-recursive. □

Consider a method $c \to \varphi$ with $|\varphi| > 1$ that is right generating, i.e., we know that c can be reached again through a non-last task (i.e., c is fully recursive). An HTN model containing such a method cannot be tail-recursive. However, if the respective partition is not left generating, the overall model can be non-self-embedding. Since every totally ordered tail-recursive HTN problem is non-self-embedding and there are recursive structures that are right generating but not self-embedding, the following holds.

Theorem 10. *The class of totally ordered non-self-embedding HTN planning problems is a strict superset of the totally ordered tail-recursive HTN planning problems.*

As result, we have found a new subclass of totally ordered HTN problems that can be translated into classical problems directly. The class is a strict super-class of the totally ordered tail-recursive problems. However, be aware that tail-recursiveness is also defined on partial-ordered problems, which is not considered here at all. Since we know that all tail-recursive HTN problems (i.e., also the partial-ordered ones) define regular languages, it is an interesting question for future work whether the presented approach can also deal with a (limited form of) partial order, or – the other way around – whether we can exploit a property similar to (non-)self-embedding also in the partial-order setting.

Be aware that our approach is not limited to non-self-embedding problems, but can be applied to all totally ordered problems. The difference is only whether approximation is needed (in self-embedding problems) or not (in non-self-embedding problems).

6. Related Work

Next we want to discuss connections of our approach to related work. We start with planning systems from the literature, especially those exploiting techniques from classical planning to solve hierarchy problems, either by adapting the techniques or by directly using them. Afterwards we discuss a technique introduced by Chrpa and Barták (2016) to provide control knowledge in classical planning, which uses a similar encoding to what we use in Section 4.5.

In classical planning there is a large number of effective domain-independent solving techniques, which makes it appealing to use them also in HTN planning. However, this is not a straightforward task due to the different expressiveness of the formalisms. Further, techniques (like heuristics) in classical planning are naturally defined based on a state-based goal definition, which is often not given in HTN planning, since it is not necessary due to the hierarchical approach.

Some hierarchical planning systems use techniques for reachability analysis as known from classical planning, e.g. Fape (Bit-Monnot et al., 2016), or Pddl4J (Pellier & Fiorino, 2021; Ramoul et al., 2017). Others use heuristics from classical planning (Lesire & Albore, 2021), enable the use of arbitrary classical heuristics (Höller et al., 2018, 2019, 2020), or adapt heuristics from classical planning to the HTN setting (Höller et al., 2020b). Besides search-based systems, there are also several solvers for HTN planning that use translations to propositional logic, e.g. Panda (Behnke et al., 2018, 2019) or Lilotane (Schreiber et al., 2019; Schreiber, 2021a, 2021b), which are based on encodings known from classical planning.

There are also approaches that – like ours – translate HTN planning problems directly into classical planning problems (Alford et al., 2009, 2016; Behnke et al., 2022). These are the approaches closest to the one presented here. Since HTN planning is more expressive than classical planning, they bound the HTN problem and translate them to a classical planning problem afterwards. The problem is then solved using a classical planning system. When no solution is found, the bound is increased. Alford et al. (2016) store the current task network in the state of the classical problem and introduce special actions simulating the application of methods by manipulating this part of the state. The used bound limits the number of tasks that can be in a task network. In general, no upper bound can be calculated (since HTN planning is undecidable). However, Alford et al. (2016) introduce a subclass called *tail-recursive* problems, for which an upper bound can be calculated. For this class, it is possible to use a single (non-incremental) translation. However, incrementing until the bound is reached works better in practice, because problems are often solvable with much smaller bounds, leading to smaller classical problems. We have seen in Section 5 that there is a connection between this class and non-self-embedding problems. Behnke et al. (2022) introduced improved variants of the translations which are much more efficient in practice. However, the basic idea is the same as before. There are two main differences between our approach and the one of Alford et al. (2016). First, the way to overcome the different expressiveness: we do not bound the problem, but approximate it. Second, the encoding itself is different: Alford et al. simulate the decomposition process in the state of a classical planning problem by using actions that simulate method application, we compile it away; our classical model only contains (extended versions of) actions from the HTN model.

Another approach on hierarchical planning, though not directly on HTN planning, has been presented by Geib and Weerasinghe (2020). The authors introduce a hierarchical planning system based on a model defined as Combinatory Categorial Grammar (Steedman, 2000), a formalism lent from natural language processing. Similar representations have been applied before in the field of plan and goal recognition (Geib, 2009; Geib & Goldman, 2009, 2011) and there are also approaches in the literature on how to learn them (Geib & Kantharaju, 2018). One motivation of Geib and Weerasinghe's work is to have a learnable, unified representation for natural language processing, plan and goal recognition, and planning. The approach on planning presented by Geib and Weerasinghe (2020) proposes

a context-free model, which makes it equally expressive as the totally ordered HTN models used in our approach (Höller et al., 2014).

Our encoding of automata in classical planning (Section 4.5) is similar to the one presented by Chrpa and Barták (2016). They introduce a language to encode control knowledge to guide a classical planning system that is similar to a finite automaton: they define a set of states and transitions that are also labeled with actions and compile it into the planning problem like we do. A solution then needs to comply with both the original classical problem and the transition system. There are two main differences to our encoding: (1) their transitions come with additional conditions that need to hold in the state where the action from the transition label is applied. These conditions might be state-based preconditions or so-called "open goals". The former are state features that must/must not be in the state, just like preconditions in classical planning. The latter are special state features that are marked in the beginning, e.g. the goal definition of the original planning problem. Taking a transition with such a condition is only permitted when the corresponding goal is still open, while it is closed when the corresponding original state feature is added for the first time. The second difference is (2) that they do not define goal states for the FA-like state transition system. Taking (1) and (2) together, their transition system cannot be interpreted on its own accord, but has to be considered together with the original planning problem, while we can consider our FA and the state-transition system defined by the actions of the planning problem as two regular languages. We are interested in their intersection.

Consequently, while Chrpa and Barták argue that resulting solutions are also solutions for the original problem (i.e., the approach is sound), they do not give a formal proof and also do not formally characterize the set of solutions to their compiled problem, which for us is necessary to prove soundness/completeness of our overall approach.

7. Implementation

Next we describe our implementation, which is available online[5]. We implemented our approach on top of the software stack of the Panda framework (Höller et al., 2021), using HDDL as input language (Höller et al., 2020a) and grounding the models using Panda's grounding system (Behnke et al., 2020).

7.1 Analysis, Approximation, and Encoding

After having obtained a ground model, we execute the actual Toad system. The implementation for the ICAPS paper (Höller, 2021) used an efficient automata representation, which however lacked flexibility of common libraries. Here we wanted to evaluate the effect of optimizations like minimization on our approach. Therefore we switched to an implementation based on the *OpenFst Library*[6]. We implemented three variants of the FA building process:

- Toad – The overall automaton is built top down as described in Algorithm 1, no optimization is applied.

5. *toad.hierarchical-task.net*
6. https://www.openfst.org

- Toad^{po} (post optimization) – The overall automaton is built top down. In a post-processing, ε transitions are removed and the automaton is determinized and minimized using the respective methods from the library.

- Toad^{io} (intermediate optimization) – The automaton is built in a bottom-up manner, and optimization is applied to the intermediate automata. This is done by constructing the decomposition graph (see Figure 3 on page 621) and creating, optimizing, and storing the automata for its tasks in a bottom-up manner. When creating the automaton for a task c, all automata for tasks reachable from c are already optimized and stored and can be looked up. Since optimization is done during the creation, this avoids to first build the entire automaton.

7.2 Solving the Classical Planning Problem

We use the Fast Downward (Fd) system to solve the resulting classical problems. In an early version of our system, we generated a PDDL output of the already ground instances and called Fd on this input including Fd's preprocessing. However, the preprocessing took very long, and since we already have a ground finite domain representation of the problem, we changed our output to Fd's intermediate model representation that is usually generated by its preprocessing. We then directly call Fd's search engine. We modified Fd such that it calls the verification procedure (see below in Section 7.3) when a solution is found. When verification fails, i.e., when it is not a solution for the underlying HTN planning problem, we continue search.

Our models can get very large and we need to use heuristics that can deal with this. We found that the h^{ff} heuristic (Hoffmann & Nebel, 2001) shows good results. We also use its preferred operators. Since it captures the hierarchy of the original problem, we assumed that the automaton has large impact on the resulting classical problems. While this information is implicitly included in the output problem, we deemed it a good idea to reuse it more explicitly for search-guidance. We created a new heuristic in Fd that returns the distance of the current automaton state to the nearest goal state of the automaton as heuristic value. We implemented this as a pre-computed heuristic, so that computation during search is very fast. We call this heuristic h^{dfad} (DFA Distance) heuristic.

Theory vs. Practice

Fd is currently the de facto standard planning system in classical planning and it also shows good performance when combined with our approach. However, in this section we illustrate a problem occurring when it is used in our overall framework.

Consider the following planning problem containing two actions $\mathcal{A} = \{a, b\}$[7] and a single abstract task $\mathcal{C} = \{A\}$. Both actions have no preconditions, b further has no effects, while

7. In principle, it should be $\{a(), b()\}$, but we omit the empty parameter lists to improve readability.

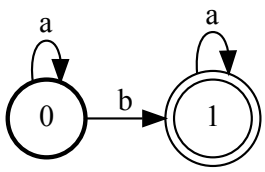

Figure 14: Automaton generated by Toad.

a fulfills the state-based goal $\{g\}$.

$$\begin{aligned} \mathit{prec}(b) &\mapsto \{\} \\ \mathit{prec}(a) &\mapsto \{\} \\ \mathit{eff}(b) &\mapsto \{\} \\ \mathit{eff}(a) &\mapsto \{g = \mathit{true}\} \end{aligned}$$

There are two methods for the abstract task A, which can either be decomposed into the action b, or into a sequence of the tasks $(a,\ A,\ a)$.

$$\begin{aligned} \mathcal{M} = \{&A \to b \\ &A \to a, A, a\} \end{aligned}$$

The hierarchy of the problem generates the context-free language $L_h = \{a^n\ b\ a^n \mid n \geq 0\}$, which we introduced earlier in Example 1 in Section 4.3 to illustrate the approximation. However, the state transition system of the problem enforces that at least one instance of a must be executed (since only the action a fulfills the state-based goal condition), i.e., $L_c = \{\omega \in \mathcal{A}^* \mid \omega \text{ contains at least one } a\}$. The language of the overall planning problem is then defined as $L_h \cap L_c = \{a^n\ b\ a^n \mid n \geq 1\}$.

When we apply the approximation presented in Section 4.3 on L_h, it results in the following set of rules:

$$\begin{aligned} \mathcal{M} = \{&A \to Y, b, Z \\ &Y \to a, Y \mid \varepsilon \\ &Z \to Z, a \mid \varepsilon\} \end{aligned}$$

They generate the regular language ${L_h}' = \{a^n\ b\ a^m \mid n, m \geq 0\}$. The corresponding finite automaton generated by Toad is shown in Figure 14.

Combined with the language of the state transition system implied by the actions, we get ${L_h}' \cap L_c = \{a^n\ b\ a^m \mid n, m \geq 0 \text{ and } n + m \geq 1\}$. When we create the classical planning problem as described in Section 4.5 and generate all solutions found by Fd, it returns only a single one, which is (b, a). However, this solution is no solution for the HTN planning problem and thus verification fails.

To illustrate why Fd returns only a single solution, we generated the entire search space of the problem using Fd. It is visualized in Figure 15. The initial state is given at the top ($\{\mathit{dfa}(s_0)\}$), the goal state at the bottom. Whether Fd returns (b, a) or (a, b) is up to tie-breaking. However, both are no solutions for the underlying HTN problem. After visiting

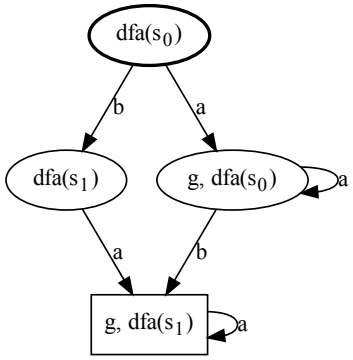

Figure 15: The entire search space generated with FD (i.e., using a graph search). Nodes represent states of the search space, arcs represent state transitions, labeled with the respective action.

the bottom state for the first time (via one of the sequences), the graph search used by FD adds the bottom state to the visited list. When reaching the bottom state again via the other sequence, it has been seen before and will thus be discarded. Further, no alternative solution is generated, since all solutions have a prefix resulting in the state $\{g, dfa(s_1)\}$. To find an HTN solution, it needs to be visited at least twice. As a result, while Theorem 5 states that our approach is complete, the FD-based implementation is incomplete, because planning systems performing a graph search do not fulfill Assumption 1 (see page 637).

To get an overall system which is complete, we need a classical planning system that eventually explores *every* solution. To make this clear: we do not need to generate all solutions *at once*, but we need that – when an approximate plan returned is not a solution for the HTN planning problem – we are able to generate the next one. For the problem given above, this would result in the set $\{a^n\ b\ a^m \mid n, m \geq 0 \text{ and } n+m \geq 1\}$, which includes the HTN solutions $\{a^n\ b\ a^n \mid n \geq 1\}$.

In FD, there is no simple option to switch from a graph to a tree search, or to implement such a search. Instead we modified the FAST FORWARD (FF) planning system (Hoffmann & Nebel, 2001) (which usually also does a graph search) to do a tree search and to return more than one solution[8]. Performing FF's A^* search using the h^{ff} heuristic, we now get the following solutions, the third one being the first one that is also a solution for the HTN planning problem.

- $\{a, b\}$
- $\{b, a\}$
- $\{a, b, a\}$
- $\{b, a, a\}$

8. Be aware the difference between the FAST FORWARD/FF *planning system* and the h^{ff} *heuristic*, which we use both in FF and in FD.

- $\{a, a, b\}$
- $\{a, b, a, a\}$
- ...

In the evaluation we use a modified Fd system which verifies solutions before returning them and continues search in the same search space (i.e., without re-starting search) when verification fails, but we do not use the Ff system afterwards. This means that our implementation is not complete. However, on the current benchmark set, this is not an issue. When it becomes an issue on future benchmark sets, starting a system like Ff used above when Fd fails will result in a complete overall system.

7.3 Verification

There are several approaches to decide whether a given sequence of actions is the solution to an HTN planning problem (HTN plan verification). They are based on a translation to propositional logic (Behnke et al., 2017), on parsing techniques (Barták et al., 2018, 2020), and on a translation to HTN planning (Höller et al., 2022). The latter generates a planning problem that has a solution if and only if the sequence is a solution to the original problem. Compared to the original problem, the compilation is much simpler to solve.

The setting most relevant for us is totally ordered HTN planning with method preconditions (a feature not supported by all of the verification systems). In combination with a progression-based planning system (we use the one of Höller et al., 2020) the translation to HTN planning had the highest coverage in a recent evaluation (see Höller et al., 2022). It reached over 99% for valid plans and over 97% for proving invalidity. Thus we use this approach to verify our plans.

8. Empirical Evaluation

This section is divided into four parts. First we investigate the impact of the applied optimizations in Section 8.1, then we compare different search configurations of Toad in Section 8.2 and have a look at unsolved instances in Section 8.3. Lastly, we compare Toad to the state of the art in HTN planning in Section 8.4. All experiments in this section ran on Intel Xeon E5-2650 CPUs with 2.30 GHz (one core per job), a time limit of 30 minutes, and a memory limit of 8 GB.

8.1 Impact of Automaton Optimization

First we compare the three variants to build the automaton: no optimization (Toad), post-optimization (Toad$^{\text{po}}$), and intermediate optimization (Toad$^{\text{io}}$). Figure 16 shows the **time in seconds** needed to build the automaton, starting *after* analysis/approximation and ending before writing the problem. In all scatter plots, marks on the axis represent instances not solved by the particular configuration (due to the time or memory limit). The left plot compares Toad and Toad$^{\text{po}}$, the right one Toad$^{\text{po}}$ and Toad$^{\text{io}}$. Each symbol represents one instance, some domains are highlighted and given in the legend. The Toad$^{\text{po}}$ configuration needs slightly longer to build the automaton than the Toad configuration

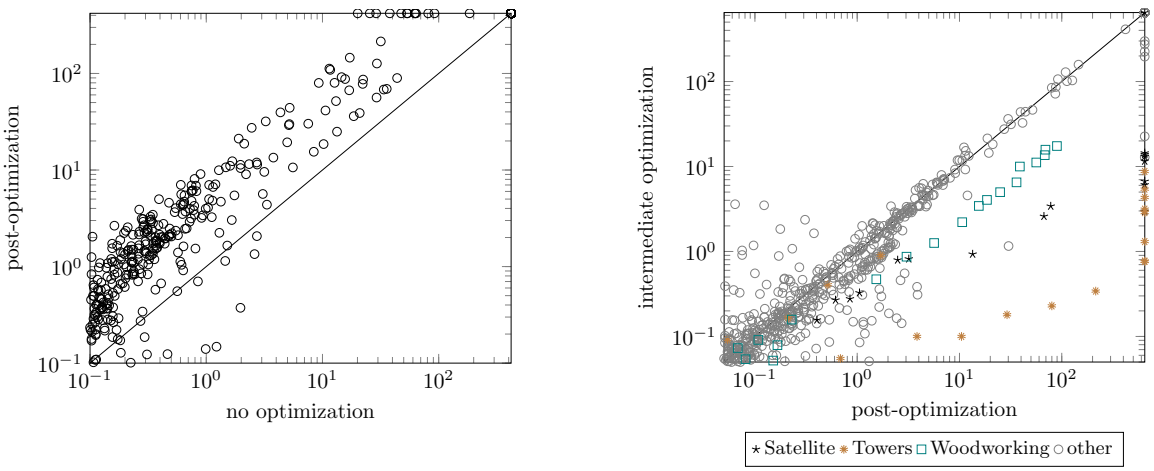

Figure 16: Time to build the automaton in seconds.

(left). Since both methods first create the automaton in exactly the same way but Toad^{po} performs optimization afterwards, this is not surprising. However, be aware that this does not mean that it does not pay off, since Figure 16 only shows the time needed to build the automaton. When we compare Toad^{po} and Toad^{io} (right), we see that the time to create the automaton does not change much in most domains, but that intermediate optimization helps in *Satellite*, *Woodworking*, and especially *Towers*.

Table 2 summarizes the impact of the optimization on the number of states of the final automaton. This number is equal for Toad^{po} and Toad^{io}, so we report the difference between using no optimization (Toad) and using optimization (Toad^{po}/Toad^{io}). For each instance, we compare the number of states (1) before optimization and (2) after optimization. We report the size of (2) in percent of (1), i.e., the 48.73 in the first row means that in the *Assembly* domain, an automaton has after optimization on average 48.48% of its original number of states. The number of states is reduced through all domains, most average reduction is achieved in *Satellite* and *Snake*. However, optimization is especially helpful in the *Towers* domain. Here the size reduction is higher on larger instances than on smaller ones (which makes the larger instances solvable).

Figure 17 gives the **overall runtime** of entire solving process of Toad in seconds. The left plot compares Toad^{po} and Toad^{io}. They behave similar in general, but Toad^{io} solves more instances in the *Towers* domain. The right-hand side of Figure 17 compares Toad and Toad^{io}. The optimization of the automaton mostly pays off, especially in the domains *BW-GTOHP*, *Elevator*, *Satellite*, *Snake*, *Towers*, *Transport*, and *Woodworking*. Only in the *Logistics* domain the performance decreases when optimization is used. This is interesting, because in this domain, the number of states is not changed much.

Next we have a look at the coverage results, i.e., the number of solved instances per domain. Table 3 compares the three optimization variants combined with greedy best first search (GBFS) and the h^{ff} heuristic. For each configuration, the absolute coverage and the coverage normalized to 1 per domain is given. Toad^{po} increases the coverage compared to Toad, and Toad^{io} performs best. Optimization has most effect in the domain *Towers*,

	μ	σ
Assembly	48.73	27.70
Barman	76.63	3.86
BW-GTOHP	12.55	20.08
BW-HPDDL	86.46	0.50
Childsnack	72.88	1.82
Depots	51.57	20.43
Elevator	11.24	15.39
Entertainment	23.73	16.68
Factories	58.27	2.60
Freecell	–	–
Hiking	66.01	15.06
Logistics	77.52	3.49
Minecraft Pl	6.83	0.00
Minecraft Reg	75.80	0.09
Monroe-FO	–	–
Monroe-PO	–	–
Multiarm-BW	85.15	0.44
Robot	82.20	16.19
Rover	40.20	24.98
Satellite	2.75	2.05
Snake	2.63	3.81
Towers	13.00	24.22
Transport	62.34	2.77
Woodworking	22.25	30.89
overall	48.48	33.25

Table 2: Size of automata after optimization in percent of their original size. The table gives the mean (μ) and the standard deviation per domain (σ).

Transport, and *Woodworking*. The increase in runtime observed in *Logistics* has no negative effect on the coverage, which is even slightly increased.

8.2 Toad Search Configurations

Next we compare different search algorithms and heuristics. The results are given in Table 4. All configurations in this comparison use intermediate optimization (Toad^{io}), i.e., the third column labeled *gbfs*(*ff*) using GBFS and the h^{ff} heuristic is the same that we already had in Table 3 (here it was the left-most column). We combine our novel h^{dfad} heuristic with h^{ff} in a configuration inspired by the Lama system (Richter & Westphal, 2010), i.e., by using a multi-queue search. The system maintains two queues, and both heuristics are computed on every search node. Nodes are inserted into both queues, one queue is sorted by h^{ff}, the other

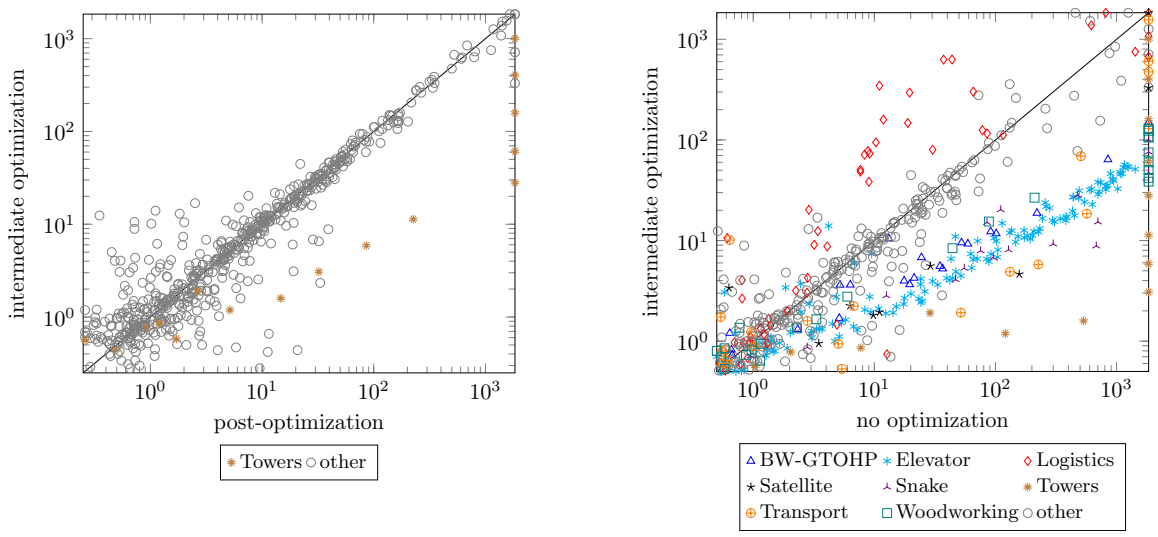

Figure 17: Total runtime of the solver in seconds.

one by h^{dfad}. The result is given in the column named $mq(dfad, ff)$. The overall coverage is again increased when compared to the GBFS/h^{ff} configuration.

Next we are interested in whether enforced hill climbing (EHC) as used by the original FF planning system yields good results. We combine it with h^{ff} (called $ehc(ff)$) and with the h^{dfad} heuristic (called $ehc(dfad)$). When compared to $mq(dfad, ff)$, we see that the latter helps in *Logistics*, *Barman*, and *Towers*. In some other domains, it yields results comparable to $mq(dfad, ff)$, but without the need to compute a heuristic (since h^{dfad} is a pre-computed heuristic).

To combine $mq(dfad, ff)$ and $ehc(dfad)$, we configured FD to do an iterated search, first performing EHC with h^{dfad} followed by the multi-queue $mq(dfad, ff)$. We found that EHC gets stuck in some domains and introduced a time limit for that search. We tried 1 minute, 2 minutes, and 5 minutes, all with similar results. In the table (and the rest of this section), the configuration with 2 minutes is used, which we denote $i[ehc, mq]$ (for iterated search). It combines the strengths of $mq(dfad, ff)$ and $ehc(dfad)$ and reaches the best performance of all our configurations.

So far we did not report any results regarding runtime. Figure 18 shows the accumulated number of solved instances over time for several TOAD configurations. Here the effect of our optimizations can be seen much better than in the coverage results. The configuration using automaton optimization solves problems much quicker than the base configuration. $mq(dfad, ff)$ performs similar to GBFS with h^{ff}, but solves a few more instances. The iterative search $i[ehc, mq]$ is very fast in the beginning, but the curve flattens between 10^1 and 10^2 seconds, before the $mq(dfad, ff)$ part of the search is started (after 120 seconds), which further solves several instances.

The next question we want to investigate is how much the (1) properties regarding recursion and (2) the size of the FA has an impact on the difficulty of the resulting classical planning problem. For this evaluation we use the $mq(dfad, ff)$ configuration. Figure 19 shows the number of states on the y axis and the search time in seconds on the x axis. The

Domain	#inst	Toad^{io}		Toad^{po}		Toad	
Assembly	30	30	1.00	30	1.00	30	1.00
Barman	20	15	0.75	15	0.75	15	0.75
BW-GTOHP	30	**23**	**0.77**	**23**	**0.77**	22	0.73
BW-HPDDL	30	21	0.70	21	0.70	21	0.70
Childsnack	30	22	0.73	22	0.73	**23**	**0.77**
Depots	30	**25**	**0.83**	24	0.80	24	0.80
Elevator	147	147	1.00	147	1.00	147	1.00
Entertainment	12	12	1.00	12	1.00	12	1.00
Factories	20	5	0.25	5	0.25	5	0.25
Freecell	60	–	–	–	–	–	–
Hiking	30	23	0.77	**24**	**0.80**	22	0.73
Logistics	80	**51**	**0.64**	**51**	**0.64**	49	0.61
Minecraft Pl	20	1	0.05	1	0.05	1	0.05
Minecraft Reg	59	39	0.66	39	0.66	39	0.66
Monroe-FO	20	–	–	–	–	–	–
Monroe-PO	20	–	–	–	–	–	–
Multiarm-BW	74	74	1.00	74	1.00	74	1.00
Robot	20	20	1.00	20	1.00	20	1.00
Rover	30	9	0.30	9	0.30	9	0.30
Satellite	20	**11**	**0.55**	9	0.45	9	0.45
Snake	20	**19**	**0.95**	**19**	**0.95**	15	0.75
Towers	20	**17**	**0.85**	12	0.60	9	0.45
Transport	40	**36**	**0.90**	**36**	**0.90**	32	0.80
Woodworking	30	**30**	**1.00**	**30**	**1.00**	20	0.67
	892	**630**	**15.70**	623	15.35	598	14.47

Table 3: Coverage of different configurations of Toad. All are using GBFS and the h^{ff} heuristic, but different FA building processes with intermediate optimization (left), post optimization (middle), and no optimization (right).

different symbols show the properties regarding recursion, i.e., whether an instance is acyclic, non-self-embedding, or self-embedding. Most interestingly, the self-embedding instances are at the top of the diagram, meaning that the instances are large with respect to the number of states, but at the same time simple to solve (with respect to search time).

8.3 Analysis of Unsolved Instances

Next we have a look at the instances Toad was not able to solve, we call such instances *unsols*. The most interesting question is which parts of the overall process causes the *unsols*. We investigate this based on our best configuration using iterative search ($i[ehc, mq]$). The results are summarized in Table 5. From left to right, the columns show the number of

Domain	#inst	$i[ehc, mq]$		$mq(dfad, ff)$		$gbfs(ff)$		$ehc(ff)$		$ehc(dfad)$	
Assembly	30	**30**	**1.00**	**30**	**1.00**	**30**	**1.00**	**30**	**1.00**	–	–
Barman	20	**17**	**0.85**	15	0.75	15	0.75	13	0.65	**17**	**0.85**
BW-GTOHP	30	**23**	**0.77**	**23**	**0.77**	**23**	**0.77**	1	0.03	1	0.03
BW-HPDDL	30	21	0.70	21	0.70	21	0.70	**22**	**0.73**	9	0.30
Childsnack	30	**24**	**0.80**	**24**	**0.80**	22	0.73	18	0.60	**24**	**0.80**
Depots	30	**26**	**0.87**	**26**	**0.87**	25	0.83	20	0.67	19	0.63
Elevator	147	**147**	**1.00**	**147**	**1.00**	**147**	**1.00**	3	0.02	**147**	**1.00**
Entertainment	12	**12**	**1.00**	**12**	**1.00**	**12**	**1.00**	7	0.58	6	0.50
Factories	20	**5**	**0.25**	**5**	**0.25**	**5**	**0.25**	–	–	**5**	**0.25**
Freecell	60	–	–	–	–	–	–	–	–	–	–
Hiking	30	**24**	**0.80**	**24**	**0.80**	23	0.77	**24**	**0.80**	–	–
Logistics	80	**80**	**1.00**	48	0.60	51	0.64	9	0.11	**80**	**1.00**
Minecraft Pl	20	1	0.05	1	0.05	1	0.05	1	0.05	1	0.05
Minecraft Reg	59	39	0.66	39	0.66	39	0.66	39	0.66	39	0.66
Monroe-FO	20	–	–	–	–	–	–	–	–	–	–
Monroe-PO	20	–	–	–	–	–	–	–	–	–	–
Multiarm-BW	74	**74**	**1.00**	**74**	**1.00**	**74**	**1.00**	70	0.95	15	0.20
Robot	20	**20**	**1.00**	**20**	**1.00**	**20**	**1.00**	**20**	**1.00**	1	0.05
Rover	30	**9**	**0.30**	**9**	**0.30**	**9**	**0.30**	2	0.07	2	0.07
Satellite	20	**20**	**1.00**	**20**	**1.00**	11	0.55	–	–	–	–
Snake	20	**19**	**0.95**	**19**	**0.95**	**19**	**0.95**	11	0.55	**19**	**0.95**
Towers	20	**18**	**0.90**	17	0.85	17	0.85	17	0.85	**18**	**0.90**
Transport	40	**40**	**1.00**	**40**	**1.00**	36	0.90	18	0.45	11	0.28
Woodworking	30	**30**	**1.00**	**30**	**1.00**	**30**	**1.00**	**30**	**1.00**	2	0.07
	892	**679**	**16.89**	644	16.34	630	15.70	355	10.77	416	8.59

Table 4: Coverage of different configurations of Toad, all using intermediate optimization for building the FA (Toad^{io}) but different search algorithms and heuristics.

instances where the process up to a given step was completed successfully: in the first one called *grounding*, it can be seen that for 41 instances, grounding was not possible[9]. Next comes the *approximation* step. Since this is not necessary for all instances, the column gives two numbers, e.g., for *BW-GTOHP*, approximation was necessary in 29 instances, and the step was completed successfully in 25 cases. The next steps are the construction of the FA, the search by Fd, and the verification (if necessary). The last column does not represent a certain step, but summarizes how many HTN instances were solved in total. The last *row* gives for every step in the Toad process the number of instances for which this particular step caused the *unsols*. Mostly this was due to grounding, approximation or the building process of the automaton, which means that once the classical planning problem is written, most instances can be solved. However, note that most of the 77 unsolved instances

9. The experiments regarding domain properties reported in Table 1 on page 623 ran with a higher memory limit, which leads to more groundable instances.

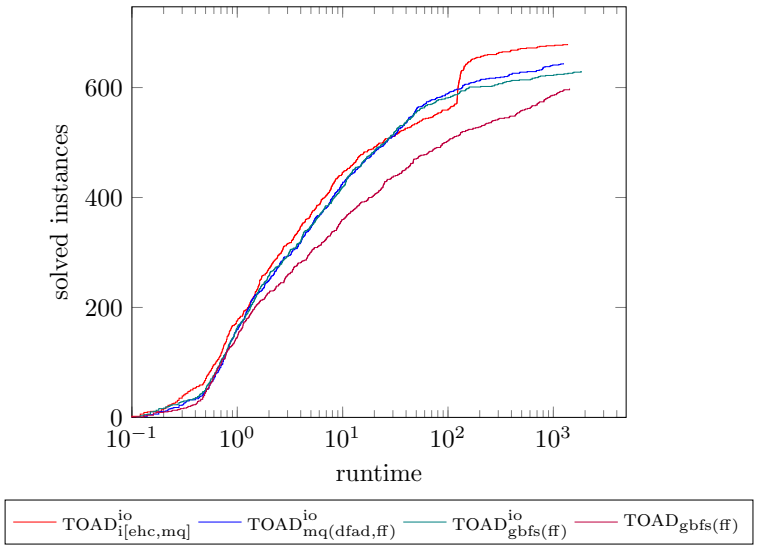

Figure 18: Accumulated number of solved instances relative to the runtime in seconds.

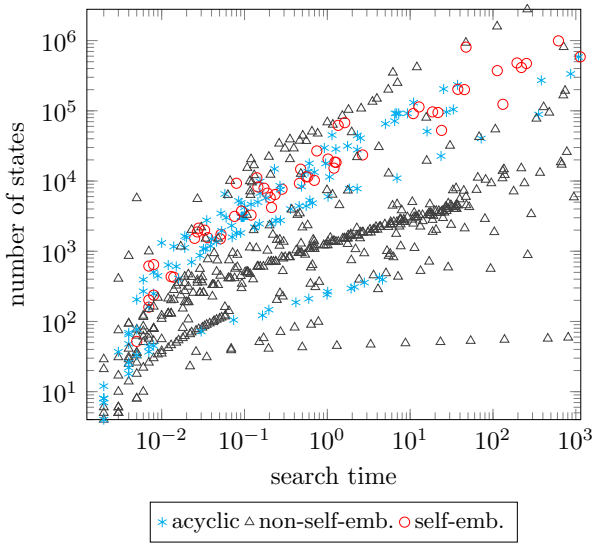

Figure 19: Automaton size against time needed for search in seconds.

reported for the approximation came from a single domain – *Freecell*. The construction of the automaton causes a high number of *unsols* in *Factories* and the two *Monroe* domains. The verification did not cause any *unsols*. Over all, when the steps before the search caused an *unsol*, this was most of the times due to the memory limit. When Fd failed to find a plan, it was mainly due to the time limit.

When we have a look at problems where approximation is needed, we find that the first solution to the classical problem is also a solution to HTN problem in most cases. This is in

		grounding	approximation	FA creation	FD search	verification	prob. solved
Assembly	30	30	–	30	30	–	30
Barman	20	20	–	**17**	17	–	17
BW-GTOHP	30	30	**25**/29	**23**	23	22/22	23
BW-HPDDL	30	30	–	30	**21**	–	21
Childsnack	30	**26**	–	**25**	**24**	–	24
Depots	30	30	**9**/10	**27**	**26**	6/6	26
Elevator	147	147	–	147	147	–	147
Entertainment	12	12	–	12	12	–	12
Factories	20	20	–	**5**	5	–	5
Freecell	60	60	**0**/60	**0**	–	–	–
Hiking	30	**26**	–	**25**	**24**	–	24
Logistics	80	80	–	80	80	–	80
Minecraft Pl	20	**4**	–	**1**	1	–	1
Minecraft Reg	59	**42**	–	**40**	**39**	–	39
Monroe-FO	20	20	20/20	**0**	–	–	–
Monroe-PO	20	20	20/20	**0**	–	–	–
Multiarm-BW	74	74	–	74	74	–	74
Robot	20	20	–	20	20	–	20
Rover	30	30	**16**/28	**9**	9	7/7	9
Satellite	20	20	20/20	20	20	20/20	20
Snake	20	20	–	**19**	19	–	19
Towers	20	20	–	20	**18**	–	18
Transport	40	40	–	40	40	–	40
Woodworking	30	30	–	30	30	–	30
		41	77	80	15	0	

Table 5: Overview of completed sub-tasks of the overall Toad process. When the number of completed instances decreased compared to the step before, it is shown bold.

line with the results reported in our conference publication (Höller, 2021), where we only had a single instance where verification failed (which was from the *Satellite* domain). Whether or not non-HTN solutions are returned further differs between the search configurations. In our $i[ehc, mq]$ configuration, this is not the case. Only three search configurations return such instances at all, all instances came from the *Satellite* domain.

8.4 Comparison to State of the Art

Lastly, we want to compare our system with the state of the art in totally ordered HTN planning. The first class of systems that we incorporate into the comparison translate the HTN problems to classical planning problems:

- Toad – Our system using EHC and multi-queue search as described above
- Htn2Strips – Translation as introduced by Alford et al. (2016)
- Htn2Sas – Translation as introduced by Behnke et al. (2022), which is based on the same approach as Htn2Strips, but includes several improvements

Further, we added a system that uses a translation to propositional logic, and two search-based systems:

- Lilotane – The runner-up of the 2020 IPC, which uses a partially grounded translation to propositional logic (Schreiber, 2021b)
- HyperTensioN – The winner of the 2020 IPC, which performs a lifted depth first search (Magnaguagno et al., 2021)
- Panda – A heuristic search-based system (Höller et al., 2018, 2020)

The coverage results are given in Table 6. Let us first consider the left part of the table, which shows the results of the systems using translations to classical planning. The Htn2Strips system shows lowest coverage, solving 146 instances less than the best system, which is Htn2Sas. However, our system is close to Htn2Sas and shows highest coverage in several domains, including *Barman*, *Depots*, *Transport*, and *Woodworking*. It is worst in the two *Monroe* domains, where it loses in total 35 instances when compared to Htn2Sas.

From the other systems, the Panda system shows highest coverage. It is on par with the Htn2Sas system and even comes with a slightly higher normalized coverage. HyperTensioN and Lilotane, the winner and runner-up of the 2020 IPC, are on par with Htn2Strips.

Figure 20 gives the accumulated number of solved instances over time. HyperTensioN reaches its highest coverage very fast, since it does not have to ground the problem, followed by Panda and Toad. The Htn2Strips is slowest, which is not surprising. In this system, the translation is done on the lifted domain and grounding is redone every time the limit bounding the HTN problem is increased. Further, the specialized HTN grounder used by Toad, Panda and Htn2Sas is much faster than Fd on the STRIPS encoding. In fact, avoiding re-grounding is one of the optimizations that Htn2Sas introduced.

To sum up, the empirical evaluation shows that our novel Toad system is competitive with the state of the art in totally ordered HTN planning and it shows higher coverage than the winner and runner-up of the 2020 IPC.

		translations to classical planning						SAT-based		search-based systems			
Domain	#inst	Toad		Htn2Sas		Htn2Str.		Lilotane		Panda		HyperT.	
Assembly	30	**30**	**1.00**	**30**	**1.00**	23	0.77	5	0.17	**30**	**1.00**	3	0.10
Barman	20	17	0.85	14	0.70	16	0.80	17	0.85	15	0.75	**20**	**1.00**
BW-GTOHP	30	23	0.77	26	0.87	21	0.70	23	0.77	**30**	**1.00**	15	0.50
BW-HPDDL	30	21	0.70	20	0.67	28	0.93	1	0.03	27	0.90	**30**	**1.00**
Childsnack	30	24	0.80	24	0.80	20	0.67	28	0.93	23	0.77	**30**	**1.00**
Depots	30	**26**	**0.87**	22	0.73	22	0.73	23	0.77	22	0.73	24	0.80
Elevator	147	**147**	**1.00**	**147**	**1.00**	107	0.73	**147**	**1.00**	**147**	**1.00**	**147**	**1.00**
Entertainment	12	**12**	**1.00**	**12**	**1.00**	4	0.33	2	0.17	**12**	**1.00**	–	–
Factories	20	5	0.25	6	0.30	6	0.30	4	0.20	**8**	**0.40**	3	0.15
Freecell	60	–	–	–	–	–	–	11	0.18	**13**	**0.22**	3	0.05
Hiking	30	24	0.80	23	0.77	24	0.80	23	0.77	**25**	**0.83**	**25**	**0.83**
Logistics	80	**80**	**1.00**	78	0.97	46	0.57	45	0.56	46	0.57	22	0.28
Minecraft Pl	20	1	0.05	1	0.05	3	0.15	4	0.20	4	0.20	**5**	**0.25**
Minecraft Reg	59	39	0.66	41	0.69	55	0.93	33	0.56	42	0.71	**57**	**0.97**
Monroe-FO	20	–	–	**20**	**1.00**	2	0.10	**20**	**1.00**	**20**	**1.00**	–	–
Monroe-PO	20	–	–	15	0.75	1	0.05	**20**	**1.00**	11	0.55	–	–
Multiarm-BW	74	**74**	**1.00**	72	0.97	73	0.99	4	0.05	**74**	**1.00**	8	0.11
Robot	20	**20**	**1.00**	**20**	**1.00**	**20**	**1.00**	11	0.55	**20**	**1.00**	**20**	**1.00**
Rover	30	9	0.30	17	0.57	9	0.30	23	0.77	29	0.97	**30**	**1.00**
Satellite	20	**20**	**1.00**	18	0.90	7	0.35	15	0.75	19	0.95	**20**	**1.00**
Snake	20	19	0.95	**20**	**1.00**	19	0.95	**20**	**1.00**	**20**	**1.00**	**20**	**1.00**
Towers	20	18	0.90	16	0.80	16	0.80	9	0.45	13	0.65	**20**	**1.00**
Transport	40	**40**	**1.00**	32	0.80	24	0.60	34	0.85	25	0.62	**40**	**1.00**
Woodworking	30	**30**	**1.00**	25	0.83	7	0.23	**30**	**1.00**	23	0.77	7	0.23
	892	679	16.89	**699**	18.18	553	13.79	552	14.58	698	**18.60**	549	14.27

Table 6: Coverage comparison of Toad and several systems from the literature.

9. Conclusion

We introduced a planning system for totally ordered HTN planning based on a translation to classical planning. Instead of bounding the problems to overcome the differences in expressiveness, as done by such systems from the literature, we use a superset approximation. The set of solutions to the classical problem is a superset of the set of solutions to the HTN problem. However, our evaluation shows that wide parts of the commonly-used benchmark set (from the 2020 IPC) can be translated without approximation. Whether or not this is necessary is decided based on a property called *self-embedding*, which is also lent from the field of formal languages and used in HTN planning for the first time. If approximation is necessary, we apply plan verification to ensure to return only valid solutions.

We show that our approach is sound and complete. Further, we show that there is a close connection of tail-recursiveness, a property used in the literature (Alford et al., 2016) to decide whether an HTN problem can directly be translated into a classical problem, and self-embedding as used here. We show that the latter describes the more general class of problems, i.e., that every totally ordered tail-recursive HTN problem is also non-self-embedding, but not vise versa.

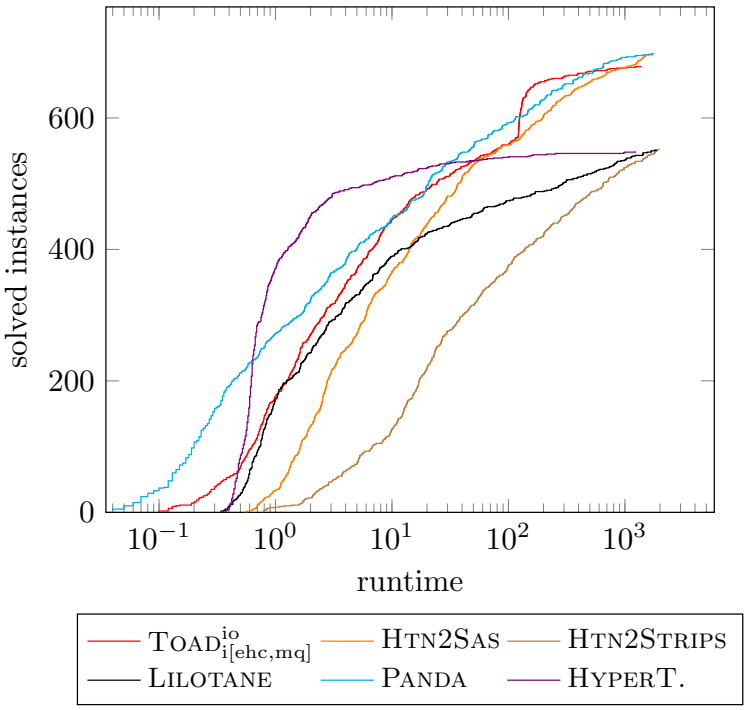

Figure 20: Accumulated number of solved instances relative to the runtime in seconds.

Our system shows good empirical results on the 2020 IPC benchmark set, where it is competitive with the search-based Panda system and the best bound-based translation system, and outperforms the systems from the 2020 IPC.

Acknowledgments.

I would like to thank the JAIR reviewers for their insightful comments and efforts towards improving the article. I further want to thank Gregor Behnke and Pascal Bercher for the discussions on the topic.

This work was funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – Project-ID 232722074 – SFB 1102 as well as by the European Union's Horizon Europe Research and Innovation program under the grant agreement TUPLES No 101070149.

Appendix A

```
(define (domain domain-htn)
   (:requirements :negative-preconditions :typing :hierarchy)
   (:types package - locatable
           location - object
           target - object
           vehicle - locatable
           locatable - object)
```

```
   (:predicates
      (road ?arg0 - location ?arg1 - location)
      (at ?arg0 - locatable ?arg1 - location)
      (in ?arg0 - package ?arg1 - vehicle))

   (:task logistics-problem :parameters ())
   (:task deliver :parameters (?p - package ?l - location))
   (:task get-to-s :parameters (?v - vehicle ?l - location))
   (:task get-to-d :parameters (?v - vehicle ?l - location))

;; self-embedding
   (:method m-deliver-ps
      :parameters ()
      :task (logistics-problem )
      :ordered-subtasks (and
       (logistics-problem )
       (logistics-problem )))

;; non-self-embedding
;  (:method m-more-ps
;     :parameters (?p - package ?l - location)
;     :task (logistics-problem )
;     :ordered-subtasks (and
;      (deliver ?p ?l)
;      (logistics-problem )))

   (:method m-one-p
      :parameters (?p - package ?l - location)
      :task (logistics-problem )
      :ordered-subtasks (deliver ?p ?l)

   (:method m-deliver
      :parameters (?l1 - location ?l2 - location ?p - package ?v - vehicle)
      :task (deliver ?p ?l2)
      :ordered-subtasks (and
       (get-to-s ?v ?l1)
       (pick-up ?v ?l1 ?p)
       (get-to-d ?v ?l2)
       (drop ?v ?l2 ?p)))

   (:method m-drive-to-s
      :parameters (?l1 - location ?l2 - location ?v - vehicle)
      :task (get-to-s ?v ?l2)
      :ordered-subtasks (drive ?v ?l1 ?l2))

   (:method m-drive-to-s-via
      :parameters (?l2 - location ?l3 - location ?v - vehicle)
      :task (get-to-s ?v ?l3)
      :ordered-subtasks (and
       (get-to-s ?v ?l2)
       (drive ?v ?l2 ?l3)))

   (:method m-drive-to-d
      :parameters (?l1 - location ?l2 - location ?v - vehicle)
      :task (get-to-d ?v ?l2)
```

```
   :ordered-subtasks (drive ?v ?l1 ?l2))

(:method m-drive-to-d-via
   :parameters (?l1 ?l2 ?l3 - location ?v - vehicle)
   :task (get-to-d ?v ?l3)
   :ordered-subtasks (and
    (drive ?v ?l1 ?l2)
    (get-to-d ?v ?l3)))

(:action drive
   :parameters (?v - vehicle ?l1 - location ?l2 - location)
   :precondition (and
         (at ?v ?l1)
         (road ?l1 ?l2))
   :effect (and
         (not (at ?v ?l1))
         (at ?v ?l2)))

(:action noop
   :parameters (?v - vehicle ?l2 - location)
   :precondition
      (and (at ?v ?l2))
   :effect ())

(:action pick-up
   :parameters (?v - vehicle ?l - location ?p - package)
   :precondition (and
         (at ?v ?l)
         (at ?p ?l))
   :effect (and
         (not (at ?p ?l))
         (in ?p ?v)))

(:action drop
   :parameters (?v - vehicle ?l - location ?p - package)
   :precondition (and
         (at ?v ?l)
         (in ?p ?v))
   :effect (and
         (not (in ?p ?v))
         (at ?p ?l))))
```

References

Alford, R., Behnke, G., Höller, D., Bercher, P., Biundo, S., & Aha, D. (2016). Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 20–28. AAAI Press.

Alford, R., Kuter, U., & Nau, D. S. (2009). Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1629–1634.

Alford, R., Shivashankar, V., Kuter, U., & Nau, D. S. (2012). HTN problem spaces: Structure, algorithms, termination. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, *11*, 625–656.

Barták, R., Maillard, A., & Cardoso, R. C. (2018). Validation of hierarchical plans via parsing of attribute grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 11–19. AAAI Press.

Barták, R., Ondrcková, S., Maillard, A., Behnke, G., & Bercher, P. (2020). A novel parsing-based approach for verification of hierarchical plans. In *Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 118–125. IEEE Computer Society.

Behnke, G., Bercher, P., Kraus, M., Schiller, M. R. G., Mickeleit, K., Häge, T., Dorna, M., Dambier, M., Manstetten, D., Minker, W., Glimm, B., & Biundo, S. (2020). New developments for Robert – Assisting novice users even better in DIY projects. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 343–347. AAAI Press.

Behnke, G., Höller, D., & Bercher, P. (Eds.). (2021). *Proceedings of the 10th International Planning Competition – Planner and Domain Abstracts*.

Behnke, G., Höller, D., & Biundo, S. (2015). On the complexity of HTN plan verification and its implications for plan recognition. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 25–33. AAAI Press.

Behnke, G., Höller, D., & Biundo, S. (2017). This is a solution! (...but is it though?) – Verifying solutions of hierarchical planning problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 20–28. AAAI Press.

Behnke, G., Höller, D., & Biundo, S. (2018). totSAT – Totally-ordered hierarchical planning through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 6110–6118. AAAI Press.

Behnke, G., Höller, D., & Biundo, S. (2019). Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 7520–7529. AAAI Press.

Behnke, G., Höller, D., Schmid, A., Bercher, P., & Biundo, S. (2020). On succinct groundings of HTN planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 9775–9784. AAAI Press.

Behnke, G., Pollitt, F., Höller, D., Bercher, P., & Alford, R. (2022). Making translations to classical planning competitive with other HTN planners. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 9687–9697. AAAI Press.

Bercher, P., Alford, R., & Höller, D. (2019). A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 6267–6275. IJCAI organization.

Bit-Monnot, A., Smith, D. E., & Do, M. (2016). Delete-free reachability analysis for temporal and hierarchical planning. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, pp. 1698–1699. IOS Press.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, *2*(2), 137–167.

Chrpa, L., & Barták, R. (2016). Guiding planning engines by transition-based domain control knowledge. In *Proceedings of the 15th International Conference Principles of Knowledge Representation and Reasoning (KR)*, pp. 545–548. AAAI Press.

Erol, K., Hendler, J. A., & Nau, D. S. (1996). Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, *18*(1), 69–93.

Geib, C. W. (2009). Delaying commitment in plan recognition using combinatory categorial grammars. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1702–1707.

Geib, C. W., & Goldman, R. P. (2009). A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, *173*(11), 1101–1132.

Geib, C. W., & Goldman, R. P. (2011). Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 958–963. AAAI Press.

Geib, C. W., & Kantharaju, P. (2018). Learning combinatory categorial grammars for plan recognition. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 3007–3014. AAAI Press.

Geib, C. W., & Weerasinghe, J. (2020). Planning using combinatory categorial grammars. In *Proceedings of the 3rd ICAPS Workshop on Hierarchical Planning (HPlan)*, pp. 18–26.

Goldman, R. P. (2009). A semantics for HTN methods. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 146–153. AAAI Press.

Goldman, R. P., & Kuter, U. (2019). Hierarchical task network planning in common Lisp: the case of SHOP3. In *Proceedings of the 12th European Lisp Symposium (ELS)*, pp. 73–80. ACM.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Höller, D. (2021). Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 159–167. AAAI Press.

Höller, D., Behnke, G., Bercher, P., & Biundo, S. (2014). Language classification of hierarchical planning problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pp. 447–452. IOS Press.

Höller, D., Behnke, G., Bercher, P., & Biundo, S. (2016). Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 158–165. AAAI Press.

Höller, D., Behnke, G., Bercher, P., & Biundo, S. (2021). The PANDA framework for hierarchical planning. *Künstliche Intelligenz (KI)*, *30*(1), 11–20.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020a). HDDL: An extension to PDDL for expressing hierarchical planning problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 9883–9891. AAAI Press.

Höller, D., Bercher, P., & Behnke, G. (2020b). Delete- and ordering-relaxation heuristics for HTN planning. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 4076–4083. IJCAI organization.

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2018). A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 114–122. AAAI Press.

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2019). On guiding search in HTN planning with classical planning heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 6171–6175. IJCAI organization.

Höller, D., Bercher, P., Behnke, G., & Biundo, S. (2020). HTN planning as heuristic progression search. *Journal of Artificial Intelligence Research*, *67*, 835–880.

Höller, D., Wichlacz, J., Bercher, P., & Behnke, G. (2022). Compiling HTN plan verification problems into HTN planning problems. In *Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 145–150. AAAI Press.

Köhn, A., Wichlacz, J., Torralba, Á., Höller, D., Hoffmann, J., & Koller, A. (2020). Generating instructions at different levels of abstraction. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*, pp. 2802–2813. International Committee on Computational Linguistics.

Lesire, C., & Albore, A. (2021). pyHiPOP – Hierarchical partial-order planner. In *Proceedings of the 2020 International Planning Competition (IPC)*.

Magnaguagno, M. C., Meneguzzi, F., & de Silva, L. (2021). HyperTensioN – A three-stage compiler for planning. In *Proceedings of the 2020 International Planning Competition (IPC)*.

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

Nederhof, M.-J. (2000a). Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, *26*(1), 17–44.

Nederhof, M.-J. (2000b). Regular approximation of CFLs: A grammatical view. In *Advances in Probabilistic and other Parsing Technologies*, chap. 12, pp. 221–241. Kluwer Academic Publishers.

Pellier, D., & Fiorino, H. (2021). Totally and partially ordered hierarchical planners in PDDL4J library. In *Proceedings of the 2020 International Planning Competition (IPC)*.

Ramoul, A., Pellier, D., Fiorino, H., & Pesty, S. (2017). Grounding of HTN planning domain. *International Journal on Artificial Intelligence Tools*, *26*(5), 1760021:1–1760021:24.

Richter, S., & Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, *39*, 127–177.

Schreiber, D. (2021a). Lifted logic for task networks: TOHTN planner Lilotane entering IPC 2020. In *Proceedings of the 2020 International Planning Competition (IPC)*.

Schreiber, D. (2021b). Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research*, *70*, 1117–1181.

Schreiber, D., Pellier, D., Fiorino, H., & Balyo, T. (2019). Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 382–390. AAAI Press.

Steedman, M. (2000). *The syntactic process.* Language, speech, and communication. MIT Press.