HDDL – A Language to Describe Hierarchical Planning Problems

D. Höller*, G. Behnke*, P. Bercher*, S. Biundo*, H. Fiorino†, D. Pellier†, and **R. Alford‡**
*Institute of Artificial Intelligence, Ulm University, 89081 Ulm, Germany
{daniel.hoeller, gregor.behnke, pascal.bercher, susanne.biundo}@uni-ulm.de
†University Grenoble Alpes, LIG, F-38000 Grenoble, France
{humbert.fiorino, damien.pellier}@imag.fr
‡The MITRE Corporation, McLean, Virginia, USA
ralford@mitre.org

Abstract

The research in hierarchical planning has made considerable progress in the last few years. Many recent systems do not rely on hand-tailored advice anymore to find solutions, but are supposed to be domain-independent systems that come with sophisticated solving techniques. In principle, this development would make the comparison between systems easier (because the domains are not tailored to a single system anymore) and – much more important – also the integration into other systems, because the modeling process is less tedious (due to the lack of advice) and there is no (or less) commitment to a certain planning system the model is created for. However, these advantages are destroyed by the lack of a common input language and feature set supported by the different systems. In this paper, we propose an extension to PDDL, the description language used in non-hierarchical planning, to the needs of hierarchical planning systems. We restrict our language to a basic feature set shared by many recent systems, give an extension of PDDL's EBNF syntax definition, and discuss our extensions, especially with respect to planner-specific input languages from related work.

1 Introduction

Much progress has been made recently in the field of hierarchical planning. Novel systems based on the traditional, search-based techniques have been introduced (Bit-Monnot, Smith, and Do 2016; Shivashankar, Alford, and Aha 2017; Bercher et al. 2017; Höller et al. 2018), but also new techniques like the translation to STRIPS/ADL (Alford, Kuter, and Nau 2009; Alford et al. 2016a), or revisited approaches like the translation to propositional logic (Behnke, Höller, and Biundo 2018a; 2018b; 2019a; 2019b; Schreiber et al. 2019). In contrast to earlier systems, such systems can be considered to be domain-independent, i.e., they do not rely on hand-tailored advice to solve planning problems, but only on their solving techniques.

Even though the systems share the basic idea of being *hierarchical* planning approaches, the feature set supported by the different systems is manifold. Bit-Monnot, Smith, and Do (2016) focus, e.g., on advanced support for temporal planning, but lack the support for recursion; several systems are restricted to models that do not include partial ordering (Alford, Kuter, and Nau 2009; Behnke, Höller, and Biundo 2018a; Schreiber et al. 2019); and some, like the one by Shivashankar, Alford, and Aha (2017) even define an entirely new type of hierarchical planning problems.

Even systems restricted to the maybe best-known and most basic hierarchical formalism, called *Hierarchical Task Network* (HTN) planning, do not share a common input language, though the differences between the input languages are sometimes rather subtle, e.g. between the formalisms used by Alford et al. (2016a) and Bercher et al. (2017). To the best of our knowledge, the hierarchical language introduced for the first International Planning Competition (McDermott et al. 1998) is not supported by any recent system.

The lack of a common language has several consequences for the field. First, it makes the comparison between the systems tedious due to the translation process. Second – and even more important – it makes the use of hierarchical planning from a practical perspective laborious, because it is not possible to model a problem at hand and try which system performs best on it. Selecting the system in beforehand (if possible) requires much insights into the systems.

A common description language would make the comparison of the systems easier, it could foster a common set of supported features and result in a common benchmark set the systems are evaluated on.

In this paper, we propose the *Hierarchical Domain Definition Language* (HDDL) as common input language for hierarchical planning problems. It is widely based on the input language of PANDA, the framework underlying the planning systems by Bercher et al. (2017), Höller et al. (2018; 2019), and Behnke, Höller, and Biundo (2018a; 2019a; 2019b). We define it as an extension of the STRIPS fragment (language level 1) of the PDDL2.1 definition (Fox and Long 2003). To concentrate on a set of features shared by many systems, we restrict the language to basic HTN planning. However, we hope that the given definition is just the starting point for further language extensions like the first PDDL version in classical planning was.

We first introduce a lifted HTN formalism from the literature, before we define our language *by example*. We go through new language elements, introduce their syntax and meaning, discuss our design choices and differences to approaches from the literature, namely PDDL1.2 (McDermott et al. 1998), SHOP(2) (Nau et al. 2003), ANML (Smith, Frank, and Cushing 2008), HPDDL (Alford et al. 2016a), GTOHP (Ramoul et al. 2017), HTN-

PDDL (González-Ferrer, Fernández-Olivares, and Castillo 2009), and HATP (de Silva, Lallement, and Alami 2015).

We then give a full EBNF syntax definition[1] based on the definition of PDDL2.1 and discuss every extension and change. We conclude with a short outlook.

2 Lifted HTN Planning

In this section we formally define the problem class HDDL can describe, i.e., standard HTN planning in line with the text book description by Ghallab, Nau, and Traverso (2004). To define the formal framework we extend the formalization of Alford, Bercher, and Aha (2015a; 2015b).

Our *lifted* formalism is based upon a quantifier-free first-order predicate logic $\mathcal{L} = (P, T, V, C)$ with the following elements. P is a finite set of *predicate symbols*, each having a finite arity. The arity defines its number of parameter variables (taken from V), each having a certain type (defined in T). Thus, T is a finite set of *type symbols* as is also known from PDDL. V is a finite set of typed variable symbols to be used by the parameters of the predicates in P. C is a finite set of typed constants. They are the syntactic representation of the objects in the real world. Please be aware that a single constant can have several types, e.g. *truck* and *vehicle* to support a type hierarchy.

The basic data structure in HTN planning is a *task network*. Task networks are partially ordered multi-sets of tasks.

In contrast to classical (non-hierarchical) planning, there are two kinds of tasks in HTN planning: primitive and compound ones. Task networks can contain both primitive tasks (also called actions) and compound tasks (also called abstract). Each task (primitive or compound) is given by its name, followed by a parameter sequence. For instance, a (primitive) task for driving from a source location $?ls$ to a destination location $?ld$ is given by the first-order atom $drive(?ls, ?ld)$. We do not differentiate between the expressions *task* and *task names* – both are used synonymously.

Definition 1 (Task Network)**.** *A task network tn over a set of* task names *X (first-order atoms) is a tuple $(I, \prec, \alpha, VC)$ with the following elements:*

1. *I is a finite (possibly empty) set of* task identifiers*.*
2. *$\prec$ is a strict partial order over I.*
3. *$\alpha : I \to X$ maps task identifiers to task names.*
4. *VC is a set of variable constraints. Each constraint can bind two task parameters to be (non-)equal and it can constrain a task parameter to be (non-)equal to a constant, or to (not) be of a certain type.*

The task identifiers are arbitrary symbols which serve as place holders (or labels) for the actual tasks they represent. We need these identifiers because any task can occur multiple times within the same task network, but the partial order needs to be able to differentiate between them. We call a task network *ground* if all task parameters are bound to (or replaced by) constants from C.

Task networks can contain primitive and/or compound tasks. *Primitive tasks* are identical to actions known from classical planning. An *action* a is a tuple $(name, pre, eff)$ with the following elements: *name* is its *task name*, i.e., a first-order atom such as $drive(?ls, ?ld)$ consisting of the (actual) name followed by a list of typed parameter variables. *pre* is its *precondition*, a first-order formula over literals over $\mathcal{L}$'s predicates. *eff* is its *effect*, a conjunction of literals over $\mathcal{L}$'s predicates (that are often divided into the positive eff^+ and the negative effects eff^-). All variables used in *pre* and *eff* are demanded to be parameters of *name*. We also write *name(a)*, *pre(a)*, and *eff(a)* to refer to these elements. We also require that for each task name *name(a)* there exists only a single action using it as its name (this way, names can be used as unique identifiers).

A *compound task* is simply a task name, i.e., an atom. In contrast to primitive tasks its purpose is not to induce a state transition, but to reference a pre-defined mapping to one or more task networks by which that compound task can be refined. They do thus not use preconditions or effects. However, there are many hierarchical planning formalisms that do also feature preconditions and/or effects for compound tasks (Bercher et al. 2016), but they are not within the scope of this paper. The before-mentioned mapping from compound tasks to pre-defined task networks is given by a set of *decomposition methods* M. A decomposition method $m \in M$ is a tuple (c, tn, VC) consisting of a compound task name c, a task network tn, and a set of variable constraints VC. The variable constraints VC allow to specify (co)designations between the parameters of c and those of the task network tn.

Definition 2 (Planning Domain)**.** *A* planning domain *$\mathcal{D}$ is a tuple $(\mathcal{L}, T_P, T_C, M)$ defined as follows.*

- *$\mathcal{L}$ is the underlying predicate logic.*
- *T_P and T_C are finite sets of primitive and compound tasks, respectively.*
- *M is a finite set of decomposition methods with compound tasks from T_C and task networks over the names $T_P \cup T_C$.*

The domain implicitly defines the set of all states S, being defined over all subsets of all ground predicates.

Definition 3 (Planning Problem)**.** *A* planning problem *$\mathcal{P}$ is a tuple $(\mathcal{D}, s_I, tn_I, g)$, where:*

- *$s_I \in S$ is the initial state, a ground conjunction of positive literals over the predicates assuming the closed world assumption.*
- *tn_I is the initial task network that may not necessarily be ground.*
- *g is the goal description, being a first-order formula over the predicates (not necessarily ground).*

HTN planning is *not* about finding courses of action achieving a certain state-based goal definition, so it makes perfect sense to specify no goal formula at all. We added them anyway to be closer to the PDDL specification. Having a goal formula in the input specification is more convenient in case one actually wants to specify a goal, it has a clearly defined semantics, and (since it can be compiled away (Geier and Bercher 2011)) causes no problems to systems that do not support it directly.

We still need to define the set of solutions for a given problem. Informally, solutions are executable, ground, primitive

[1] Syntax definitions for the ANTLR and Bison parser generators can be found online at www.uni-ulm.de/en/in/ki/panda.

task networks that can be obtained from the problem's initial task network via applying decomposition methods, adding ordering constraints, and grounding.

Lifted problems are a compact representation of their ground instantiations that are, as in classical planning, up to exponentially smaller (Alford, Bercher, and Aha 2015a; 2015b). However, we define solutions based on their grounding. The semantics of such a lifted problem is thus defined in terms of the standard semantics of its ground instantiation. We assume that the reader is familiar with the grounding process and refer to the paper by Alford, Bercher, and Aha (2015a) for details about it. To the best of our knowledge there are currently only two publications devoted to grounding in more detail – by Ramoul et al. (2017)[2] and by Behnke et al. (2019b). We now give the required definitions based on a *ground* problem and domain. Note that we do not need to represent variable constraints anymore since their constraints are already represented within the groundings.

Given ground problems/models we can now define *executability* of task networks. Let A be the set of ground actions obtained from T_P. An action $a \in A$ is called executable in a state $s \in S$ if and only if $s \models pre(a)$. The state transition function $\gamma : S \times A \to S$ is defined as in classical planning: If a is executable in s, then $\gamma(s,a) = (s \setminus \mathit{eff}^-(a)) \cup \mathit{eff}^+(a)$, otherwise $\gamma(s,a)$ is undefined. The extension of γ to action sequences, $\gamma^* : S \times A^* \to S$ is defined straightforwardly.

Definition 4 (Executability)**.** *A task network $tn = (I, \prec, \alpha)$ is called executable if and only if there is a linearization of its task identifiers $i_1, \dots, i_n$, $n = |I|$, such that $\alpha(i_1), \dots, \alpha(i_n)$ is executable in s_I.*

The means of transforming one task network into another to obtain executable task networks is decomposition.

Definition 5 (Decomposition)**.** *Let $m = (c, (I_m, \prec_m, \alpha_m))$ be a decomposition method, $tn_1 = (I_1, \prec_1, \alpha_1)$ a task network, and $I_m \cap I_1 = \emptyset$ (the latter can be achieved by renaming). Then, m decomposes a task identifier $i \in I_1$ into a task network $tn_2 = (I_2, \prec_2, \alpha_2)$ if and only if $\alpha_1(i) = c$ and*

$$
\begin{aligned}
I_2 &= (I_1 \setminus \{i\}) \cup I_m \\
\prec_2 &= (\prec_1 \cup \prec_m \cup \\
&\qquad \{(i_1, i_2) \in I_1 \times I_m \mid (i_1, i) \in \prec_1\} \cup \\
&\qquad \{(i_1, i_2) \in I_m \times I_1 \mid (i, i_2) \in \prec_1\}) \\
&\qquad \setminus \{(i', i'') \in I_1 \times I_1 \mid i' = i \text{ or } i'' = i\} \\
\alpha_2 &= (\alpha_1 \cup \alpha_m) \setminus \{(i, c)\}
\end{aligned}
$$

Now we can formally define the solution criteria.

Definition 6 (Solutions)**.** *Let $\mathcal{P} = (\mathcal{D}, s_I, tn_I, g)$ be a planning problem with $\mathcal{D} = (\mathcal{L}, T_P, T_C, M)$ and $tn_S = (I_S, \prec_S, \alpha_S)$. tn_S is a* solution to an HTN planning problem P *if and only if*

- *There is a sequence of decompositions from tn_I to $tn = (I, \prec, \alpha)$, such that $I = I_S$, $\prec \subseteq \prec_S$, and $\alpha = \alpha_S$*
- *tn_S is primitive and has an executable action linearization leading to a state $s \models g$.*

[2]Their procedure allows to delete effectless actions (Ramoul et al. 2017), which is not allowed in standard HTN planning and would e.g. invalidate the compilation for goal descriptions.

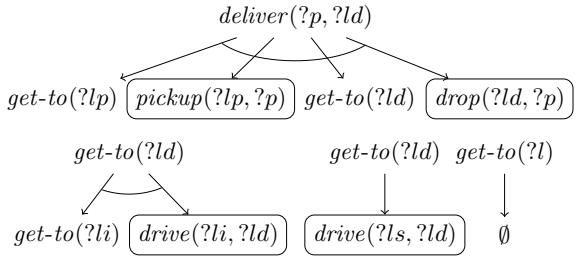

Figure 1: The method set of a simple transport domain. Actions are given as boxed nodes, abstract tasks are unboxed. All methods are totally ordered. There exists a smaller, equivalent model. However, the model has been created this way to illustrate the different language features.

3 HDDL by Example

In this section we explain our extensions to the PDDL definition based on a transport domain. To keep the example simple, the domain includes only a single transporter that has to deliver one or more packages. For each new language element we introduce its syntax and meaning and discuss the way it is modeled in other input languages.

The predicate and type definition is the same as in PDDL:

```
(define (domain transport)
  (:types location package - object)
  (:predicates
     (road ?l1 ?l2 - location)
     ...)
```

All other languages except for HATP (de Silva, Lallement, and Alami 2015) use the same theoretical model of objects and predicates as PDDL. HATP models its objects in an object-oriented way instead and further allows for SAS+ variables (Bäckström and Nebel 1995) in the input language.

The full method set of the domain is illustrated in Figure 1. Each method will be discussed in this section.

The domain contains two abstract tasks *deliver* and *get-to*. We propose to include an explicit definition of abstract tasks as it is the case for actions. HPDDL (Alford et al. 2016a) also defines abstract tasks explicitly, albeit with a slightly different syntax. Both ANML (Smith, Frank, and Cushing 2008) and HTN-PDDL (González-Ferrer, Fernández-Olivares, and Castillo 2009) require an explicit declaration of abstract tasks and their parameter types as well, but here the declaration is not separated from other elements of the domain as both declare methods together with their abstract tasks.

Some description languages for HTN problems define abstract tasks only in an implicit way by their use in methods. This includes the language used by SHOP and SHOP2 (Nau et al. 2003), PDDL1.2 (McDermott et al. 1998), HATP, as well as GTOHP (Ramoul et al. 2017). SHOP and GTOHP assume that any task that is used in a method, but is not declared to be an action is an abstract task. In contrast,

PDDL1.2 assumes that every task that has no methods is primitive. This way of implicitly defining the set of compound tasks has also been chosen in some formal definitions of hierarchical problem classes (Alford, Bercher, and Aha 2015a; 2015b). However, this can be very cumbersome when debugging domains. If the modeler forgot to define a specific primitive task, the domain will still be valid, as it would be interpreted as an abstract task.

Another problem with such a definition is that the argument types are defined implicitly, namely as those with which the task can be instantiated via any method. The language of GTOHP further does not allow for using different types (that share a common ancestor in the type hierarchy) to be used for the same task. For example, there might be different methods for the $deliver$ task, depending on the type of transported package. $deliver$ might have two methods, one where the first argument is of type $regularPackage$ and one where it is of type $valuablePackage$, the latter requiring an armored transporter. We assume that $regularPackage$ and $valuablePackage$ are disjunct types, but have a common super-type $package$, which would be the correct parameter type for $deliver$'s first argument. If its type is not declared explicitly, the planner can either reject the domain, as GTOHP does, or would have to infer the possible types of the arguments of an abstract task.

Declaring abstract tasks and their parameter types explicitly is also in line with the design choices of PDDL. Similar to abstract tasks, PDDL could omit the explicit definition of predicates as their types could be inferred from their usages. This is however discouraged from a modeling point-of-view.

Omitting the distinct definition of tasks and methods would also mean a significant deviation from the contemporary theoretical work on HTN planning. It could hinder further language extensions like annotating abstract tasks with constraints, e.g. preconditions and effects, as done by a couple of systems (see e.g. the survey by Bercher et al., 2016).

Here is the abstract task definition for the example:

```
6   (:task deliver :parameters (?p - package
        ?l - location))
7   (:task get-to :parameters (?l - location))
```

There is only a single method in the model to decompose deliver tasks (given at the top of Figure 1). It decomposes the task into four ordered sub-tasks: getting to the package, picking it up, getting to its final position, and dropping the package. The definition in HDDL could look like this:

```
8   (:method m-deliver
9     :parameters (?p - package
          ?lp ?ld - location)
10    :task (deliver ?p ?ld)
11    :ordered-subtasks (and
12      (get-to ?lp)
13      (pick-up ?ld ?p)
14      (get-to ?ld)
15      (drop ?ld ?p)))
```

The method definition starts with the name of the method that can e.g. be used to describe the decompositions needed to find a solution. We decided to give the method's parameters explicitly (line 9). This allows e.g. to restrict the types used in the subtasks and the decomposed task to subtypes of the original task parameters. Similarly, we can restrict the method to be applicable only to certain parameters of the abstract task it decomposes. To be correctly defined, we assume these parameters to be a superset of all parameters used in the entire method definition. The parameter definition is followed by the specification of the abstract task decomposed by the method as well as its parameters (line 10).

The same syntactical structure is used by HPDDL. In contrast, ANML, PDDL1.2, HATP and HTN-PDDL aggregate all decomposition methods belonging to a single abstract task, which have to be declared as part of the definition of an abstract task. As such, the variables that are declared as the arguments of an abstract task are automatically variables in a methods' task network. All of them type variables in methods explicitly.

In GTOHP's language, methods don't have names, but are identified via the abstract task they refine.

In SHOP, all variables inside a method are only defined implicitly by their usage as parameters of tasks and predicates inside the method. For example, the definition of a SHOP method starts with `:method` followed by an abstract task and its parameters – which if they are variables are automatically declared as new (untyped) variables. The same holds for variables that only occur as parameters of a method's subtasks. GTOHP and HTN-PDDL follow this pattern, but enforce that the parameters of the abstract task are typed, i.e., declared explicitly. Their languages however do not allow to specify the types of variables that occur in the method that are not parameters of the abstract task. Declaring the variables is, again, in line with the PDDL standard and e.g. done the same way in actions. We think it less error-prone. When the modeler explicitly defines the variables and their types, the system can check the compatibility of types and warn the modeler when undeclared variables are used (e.g. due to a spelling error).

The subtasks of the method are given afterwards (starting in line 11). We decided to have two keywords to start the definition `:ordered-subtasks` (as given here) and `:subtasks` (which we will show in the next method). When the `:ordered-subtasks` keyword is used, the given list of subtasks is supposed to be totally ordered. HPDDL uses the keyword `:tasks`, which might cause errors if mixed up with the `:task` keyword. Since GTOHP does only support totally-ordered HTN planning problems, their language only allows for specifying sequences of actions with the keyword `:expansion`.

In the subtask section, all abstract tasks and actions defined in the domain can be used as subtasks (and only these). The variables defined in the method's parameter section and the constants defined in the domain may be used as parameters (and only these).

The $get\text{-}to$ task from our example domain is again abstract and may be decomposed by using one of the three methods given at the bottom of Figure 1. We start with the left one that is used when there is no direct road connection. Then the transporter needs to go to the final location $?ld$ via some intermediate location $?li$. Therefore the method decomposes the task into another abstract $get\text{-}to$ task, fol-

lowed by a *drive* action with the destination location $?ld$.

```
16  (:method m-drive-to-via
17    :parameters (?li ?ld - location)
18    :task (get-to ?ld)
19    :subtasks (and
20      (t1 (get-to ?li))
21      (t2 (drive ?li ?ld)))
22    :ordering (and
23      (t1 < t2)))
```

Line 19 shows the aforementioned `:subtask` definition that allows for partially ordered tasks. The task definition contains IDs that can be used to define ordering constraints (line 22). They consist of a list of individual ordering constraints between subtasks. However, in the given example the resulting ordering is, again, a total order (and is just defined that way to demonstrate this kind of definition).

HPDDL uses the same keyword, but with a slightly different syntax so specify ordering constraints. The format omits the `and` and the `<` signs. We would argue that our notation is better readable to humans. As stated above, GTOHP cannot specify partial orders. ANML is primarily designed for temporal domains and uses a temporal syntax, e.g. `end(t1) < start(t2)`. SHOP2 and HTN-PDDL use a different approach to represent the task ordering. Instead of specifying individual ordering constraints, they require to specify the order as a single expression. This expression is a nested definition of the ordering, which can only contain two constructors: `ordered` and `unordered`. In SHOP2, e.g. `((:unordered (t1 t2) t3) t4)` corresponds to the ordering constraints `t1 < t2`, `t2 < t4`, and `t3 < t4`. Note that this construction cannot express all possible partially-ordered sets of tasks. Consider an ordering over five task identifiers `t1`, ..., `t5`, where `t1 < t4`, `t2 < t4`, `t2 < t5`, and `t3 < t5`. This ordering cannot be expressed with SHOP's nested ordered/unordered constructs. PDDL1.2 also uses this mode as a default, but does with an additional requirement also allow for an order specification as we and HPDDL do. Notably PDDL1.2 intertwines the definition of a method's subtasks and the definition of their order. The syntax of PDDL1.2 to specify the contents of methods and the order of tasks in them is somewhat convoluted and not easily readable. Thus, we have not adapted their syntax.

HATP uses a programming-language-style syntax for the encoding of methods. It further provides explicit means to determine the order in which groundings of methods should be explored during progression search. HATP's syntax for methods allows for specifying partial order, but its semantics is different from standard HTN planning. A HATP method containing partial order is interpreted as multiple totally-ordered method, one for each linearization of the given partial order. This allows for a more compact representation, but prohibits task interleaving.

HDDL – as HPDDL, SHOP2, HTN-PDDL, and ANML – only allow to specify a fixed *set* of ordering constraints. Notably, the HTN planner UMCP (Erol, Hendler, and Nau 1994) allows for *arbitrary formulae* that specify these orderings. E.g. they allow to specify an ordering $(t_1 \prec t_2) \implies (t_3 \prec t_4)$. We have not included such a generic means to formulate ordering constraints into HDDL as they do not seem to be used and supported by any current HTN planning system. In principle however, HDDL could be extended to support such complex ordering constraints.

A common feature of many HTN planning systems is the possibility of specifying state-based preconditions for methods as supported by the SHOP2 system. The feature is somewhat problematic. First, because it is (at least from our experience) usually used to guide the search and thus often breaks with the philosophy of PDDL to specify a model that does not include advice. The second problem is the way it is usually realized in the HTN planning systems: The systems introduce a new primitive task that holds the method's preconditions. It is added to the method and placed before all other tasks in the method's subtask network. Consider a totally ordered domain (i.e., the subtasks of all methods and the initial task network are totally ordered): here, the action is executed directly before the other subtasks of the method and the position where the preconditions are checked is fine. Now consider a partially ordered domain: here, the newly introduced action is not necessarily placed directly before the other subtasks, but we just know that it is placed somewhere before, i.e., the condition did hold at some point before the other tasks are executed, but may have changed meanwhile. However, though we are aware of these problems, the feature is often used and thus we integrated it and assume the standard semantics as given above.

The preconditions are defined as follows:

```
24  (:method m-already-there
25    :parameters (?l - location)
26    :task (get-to ?l)
27    :precondition (tAt ?l)
28    :subtasks ())
```

Here the method may be applied in a state where the transporter is already located at its destination. The given method has therefore no subtasks, but still has to assure that the transporter is at its destination.

Method preconditions are typically featured in languages expressing HTNs. HPDDL uses the same syntax we are proposing. GTOHP uses, as noted above, a separate `:constraints` section, where the method precondition has to be specified as a `before` constraint. This is (presumably) to allow for other state constraints later on. PDDL1.2 also features method preconditions, but they are specified as part of the task network. In ANML, there is no explicit means for writing down method preconditions, but they can be encoded into the state constraints allowed by ANML.

There is a strong contrast between what can be expressed in SHOP[3] and all other HTN formats. In SHOP, several methods for the same abstract task can be arranged in a single method declaration, each featuring its own method precondition. For the i^{th} method to be usable, it is not sufficient that its precondition is satisfied. In addition, the preconditions of all previous methods have to be not satisfied

[3]This potentially also applies to HTN-PDDL, as they use a similar syntax. Their description is unfortunately not explicit on the critical point in semantics (González-Ferrer, Fernández-Olivares, and Castillo 2009).

as well. Thus SHOP's method preconditions are in essence a chain of if-else constructs. This structure can be compiled into several individual methods with preconditions. In case one of the preconditions contains an existential quantifier (or in SHOP's case a free variable) this leads to universal quantified preconditions in the methods after it. Nevertheless we propose to drop the ability to use such if-else chains, most notably, since none of the newer languages supports it. Further, this kind of if-else is essentially a means to guide a depth-first search planner in an efficient way. Thus it does not constitute physics of the domain, but advice to the planner, which should not be part of the domain description language for a domain-independent planner.

In addition to method preconditions, HPDDL features method effects, which are modeled after SHOP2's assert and retract functionality. Method effects are executed in the state in which the method preconditions are evaluated. As far as we know, their formal semantics is not defined in any publication. We propose to drop this feature (at least for the given definition intended to be the core language). It is not commonly used and might be difficult to use for newcomers to HTN planning. Note that even without method effects in the description language, we can still simulate them with additional actions in the methods' definitions.

Sometimes it might be useful to define constraints in a method, e.g. on its variables or sorts. This is demonstrated in the following example where the transporter's source position must be different from its destination.

```
29  (:method m-direct
30    :parameters (?ls ?ld - location)
31    :task (get-to ?ld)
32    :constraints
33      (not (= ?li ?ld))
34    :subtasks (drive ?ls ?ld))
```

We are aware that PDDL allows for variable constraints in the precondition of actions. Due to consistency we also argue to allow this when method preconditions are specified. However, many HTN models are defined without methods that have preconditions and we think it not intuitive to specify a `precondition` section solely to define variable constraints. Furthermore, we think that other constraints apart from simple variable constraints might be added to the standard. These might, e.g., be constants that certain state features must hold between two tasks, or directly before some task. Therefore we integrated a constraint section to the method definition (line 32f) though our current definition only allows for equality and inequality constraints.

HPDDL places the variable constraints of a method into the method's preconditions. In addition to equality and inequality it features type constraints, where e.g. `(valuablePackage ?p)` is the constraint that `?p` belongs to the type `valuablePackage`. GTOHP allows for equality and inequality constraints that are also within the `:constraints` section, but are located in a separate `before` block. In SHOP's syntax, variable constraints have to be compiled into method preconditions referring to predicates for the individual types and an explicitly declared `equals` predicate. ANML also allows for variable constraints that can be declared freely inside a method.

We left the action definition unchanged compared to the PDDL standard we build on. Therefore we included only the following action into our example.

```
35  (:action drive
36    :parameters (?l1 ?l2 - location)
37    :precondition (and
38      (tAt ?l1)
39      (road ?l1 ?l2))
40    :effect (and
41      (not (tAt ?l1))
42      (tAt ?l2)))
43  ...)
```

The problem file is slightly adapted to represent the additional elements necessary for HTN panning (line 6).

```
1 (define (problem p)
2  (:domain transport)
3  (:objects
4   city-loc-0 city-loc-1 city-loc-2 -
        location
5   package-0 package-1 - package)
6  (:htn
7   :tasks (and
8    (deliver package-0 city-loc-0)
9    (deliver package-1 city-loc-2))
10  :ordering ()
11  :constraints ())
12 (:init
13  (road city-loc-0 city-loc-1)
14  (road city-loc-1 city-loc-0)
15  (road city-loc-1 city-loc-2)
16  (road city-loc-2 city-loc-1)
17  (at package-0 city-loc-1)
18  (at package-1 city-loc-1)))
```

The section starts with a keyword that specifies the problem class. In this example, it starts with `:htn` to define a standard HTN planning problem. However, there are several other problem classes in hierarchical planning. An example for such a class is HTN planning with task insertion, where the planner is allowed to insert tasks apart from the hierarchy. An overview of hierarchical problem classes can be found in the survey by Bercher, Alford, and Höller (2019). Some of the described problem classes are even syntactically equivalent to standard HTN planning problems and only differ in their solution criteria. By making the specification of the problem class explicit, extensions to the language can easily add new classes.

The definition of the initial task network is nested in this section. It has the same form as the methods' subtask networks. The other description languages for HTN planning also allow for a similar definition of the initial plan. Again, all of them use a slightly different syntax to describe them.

In the given example, the planning process is started with two *deliver* tasks, one for each package. These initial tasks are not ordered with respect to each other, i.e., their subtasks may be executed interleaved.

In the original PDDL standard, the domain designer has to specify a state-based goal. HTN planning problems do not require such a goal and thus often do not specify one. Therefore we made its definition optional.

4 Full Syntax Definition

We defined our syntax as close as possible to the STRIPS part (i.e., language level 1) of the PDDL 2.1 language definition of Fox and Long (2003). Wide parts of the following definition are **identical** to their definition. Changes and extensions are discussed in the following.

The domain definition has been extended by definitions for compound tasks (line 6) and methods (line 7).

```
1 <domain> ::= (define (domain <name>)
2     [<require-def>]
3     [<types-def>]:typing
4     [<constants-def>]
5     [<predicates-def>]
6     <comp-task-def>*
7     <method-def>*
8     <action-def>*)
```

The definition of the basic elements is nearly unchanged.

```
9  <require-def> ::=
       (:requirements <require-key>+)
10 <require-key> ::= ...
11 <types-def> ::= (:types <types>+)
12 <types> ::= <typed list (name)>
       | <base-type>
13 <base-type> ::= <name>
14 <constants-def> ::=
       (:constants <typed list (name)>)
15 <predicates-def> ::=
       (:predicates <atomic-formula-skeleton>+)
16 <atomic-formula-skeleton> ::=
       (<predicate> <typed list (variable)>)
17 <predicate> ::= <name>
18 <variable> ::= ?<name>
19 <typed list (x)> ::= x+ - <type>
       [<typed list (x)>]
20 <primitive-type> ::= <name>
21 <type> ::= (either <primitive-type>+)
22 <type> ::= <primitive-type>
```

The only change concerns the definition of `<types-def>` (lines 11 and 13) in combination with the definition of `<typed list (name)>` (line 19). In the PDDL2.1 standard, this can be realized by a list of names, e.g. in an untyped way. Our intention was to enforce a typed model and therefore allow for untyped elements only in the type definition. There, it is necessary to define the base type(s). In every other definition that includes `<typed list (name)>` (e.g. parameter and constant definitions), we wanted to enforce a typed list.

Abstract tasks are defined similar to actions.

```
23 <comp-task-def> ::= (:task <task-def>)
24 <task-def> ::= <task-symbol>
       :parameters (<typed list (variable)>)
25 <task-symbol> ::= <name>
```

In a standard HTN setting, methods consist of a parameter list (line 27), the abstract task they decompose (line 28), and the resulting task network (line 30). The parameters of a method are supposed to include all parameters of the abstract task that it decomposes and those of the tasks in its network of subtasks.

By setting the `:htn-method-prec` requirement, one might use method preconditions (line 29).

```
26 <method-def> ::= (:method <name>
27     :parameters (<typed list (variable)>)
28     :task (<task-symbol> <term>*)
29     [:precondition <gd>]:htn-method-prec
30     <tasknetwork-def>)
```

The definition of task networks is used in method definitions as well as in the problem definition to define the initial task network. It contains the definition of sub-tasks (line 32), ordering constraints (line 33), and variable constraints (line 34) between any method parameters.

When the key `:ordered-subtasks` is used, the network is regarded to be totally ordered. In the other cases, ordering relations may be defined explicitly. This is done by including ids into the task definition that can then be referenced in the ordering definition.

```
31 <tasknetwork-def> ::=
32     [:[ordered-][sub]tasks
           <subtask-defs>]
33     [:order[ing] <ordering-defs>]
34     [:constraints <constraint-defs>]
```

We use the same syntax definition for method subnetworks and the initial task network. Here, the keyword `subtasks` would seem odd. Therefore the syntax also allows for the keys `tasks` and `ordered-tasks` (line 32) that are supported to be used in the initial task network.

The subtask definition may contain one or more subtasks. A single task consists of a task symbol and a list of parameters. In case of a method's subnetwork, these parameters have to be included in the method's parameters, in case of the initial task network, they have to be defined as constants in s_0 or in a dedicated parameter list (see definition of the initial task network, line 82). The tasks may start with an id that can be used to define ordering constraints.

```
35 <subtask-defs> ::= () | <subtask-def>
       | (and <subtask-def>+)
36 <subtask-def> ::= (<task-symbol> <term>*)
       | (<subtask-id> (<task-symbol> <term>*))
37 <subtask-id> ::= <name>
```

The ordering constraints are defined via the task ids. They have to induce a partial order.

```
38 <ordering-defs> ::= () | <ordering-def>
       | (and <ordering-def>+)
39 <ordering-def> ::=
       (<subtask-id> "<" <subtask-id>)
```

So far we only included variable constraints into the constant section, but the definition might be extended in further language levels, of course.

```
40 <constraint-defs> ::= () | <constraint-def>
       | (and <constraint-def>+)
41 <constraint-def> ::= ()
       | (not (= <term> <term>))
       | (= <term> <term>)
```

The original action definition of PDDL has been split to reuse its body in the task definition.

```
<action-def> ::= (:action <task-def>
    [:precondition <gd>]
    [:effects <effect>])
```

We restricted the definition of preconditions and effects to level 1, i.e., the STRIPS part of the overall language.

```
<gd> ::= ()
<gd> ::= <atomic formula (term)>
<gd> ::=:negative-preconditions <literal (term)>
<gd> ::= (and <gd>*)
<gd> ::=:disjunctive-preconditions (or <gd>*)
<gd> ::=:disjunctive-preconditions (not <gd>)
<gd> ::=:disjunctive-preconditions (imply <gd> <gd>)
<gd> ::=:existential-preconditions
    (exists (<typed list (variable)>*) <gd>)
<gd> ::=:universal-preconditions
    (forall (<typed list (variable)>*) <gd>)
<gd> ::= (= <term> <term>)
<literal (t)> ::= <atomic formula(t)>
<literal (t)> ::= (not <atomic formula(t)>)
<atomic formula(t)> ::= (<predicate> t*)
<term> ::= <name>
<term> ::= <variable>
<effect> ::= ()
<effect> ::= (and <c-effect>*)
<effect> ::= <c-effect>
<c-effect> ::=:conditional-effects
    (forall (<variable>*) <effect>)
<c-effect> ::=:conditional-effects
    (when <gd> <cond-effect>)
<c-effect> ::= <p-effect>
<p-effect> ::= (not <atomic formula(term)>)
<p-effect> ::= <atomic formula(term)>
<cond-effect> ::= (and <p-effect>*)
<cond-effect> ::= <p-effect>
```

The problem definition includes as additional element the initial task network (line 74). Since a state-based goal definition is often not included in HTN planning, we made it optional (line 76).

```
<problem> ::= (define (problem <name>)
    (:domain <name>)
    [<require-def>]
    [<p-object-declaration>]
    [<p-htn>]
    <p-init>
    [<p-goal>])
<p-object-declaration> ::=
    (:objects <typed list (name)>)
<p-init> ::= (:init <init-el>*)
<init-el> ::= <literal (name)>
<p-goal> ::= (:goal <gd>)
```

The initial task network contains the definition of the problem class (line 81). In this first definition we only included standard HTN planning.

```
<p-htn> ::= (<p-class>
    [:parameters (<typed list (variable)>)]
    <tasknetwork-def>)
<p-class> ::= :htn
```

Our overall definition includes two new requirement flags:

- `:htn` requires the applied system needs to support HTN planning at all, so this can be seen as the basic requirement for the language defined here.
- `:htn-method-prec` requires the applied system needs to support method preconditions.

5 Discussion

We consider the language proposed in this paper as a first step towards a standardized language for hierarchical planning problems and hope that it helps to find a minimal set of features supported by the diverse systems. However, this basic feature set as well as many design options are still open and have to be discussed in the research community.

First of all, we think it is important to remain as close as possible to PDDL and to reuse its features to allow domain modelers to create both hierarchical and non-hierarchical problems with minimal learning effort. Then, we must decide which features have to be at the core of the language, and which ones are secondary and possibly could be ignored. This is especially important to establish a competition to compare the performance of different systems (see the proposal by Behnke et al. (2019a)).

A feature that was present in the early HTN formalisms (see e.g. the formalism by Erol, Hendler, and Nau, (1994)) is the possibility to define more elaborated constraints in task networks. Recent work in hierarchical planning was not based on such a rich definition language, but on rather minimalistic formalisms like the one introduced by Geier and Bercher, (2011). In this first definition we only included the very basic constraints: ordering constraints, variable constraints, and method preconditions. However, we think that a constraint set as given in PDDL3 might be a nice extension beneficial for domain designers. When the community wants to foster application in real world domains, it may be necessary to integrate support for numbers and time into the planning systems. Since our definition builds upon the PDDL2.1, at least the extension of the syntax in that direction could easily be done. Another possible extension is the support for preconditions and effects in the definition of abstract tasks (see Bercher et al. (2016) for an overview of that feature).

Beside new features, it might be interesting to include new problem classes like *HTN planning with task insertion*, *decompositional planning*, or *HGN planning*, which comes with the ability to decompose not tasks, but also goals (Shivashankar et al. 2012) and that even has been combined with task decomposition (Alford et al. 2016b).

6 Conclusion

We propose a common description language for hierarchical planning problems. We argue that the core feature set underlying many hierarchical planners from the last years is that of HTN planning and introduced its elements as an extension of PDDL. We defined the language in a way that can easily be extended by further features as has been done in PDDL. We introduced our novel language elements "by example" and discussed our design choices, the syntax used in related work, and the proposed meaning. We gave a full syntax definition afterwards and discussed the extensions and changes

to the PDDL standard. We hope that a common input language may foster the cooperation between groups working in hierarchical planning, the comparison of different hierarchical planning systems, and the application on real problems, because it enables an easy exchange of the planning system used for a given problem.

Acknowledgements

This work was partly funded by the technology transfer project "Do it yourself, but not alone: Companion-Technology for DIY support" of the SFB/TRR 62 funded by the German Research Foundation (DFG). The industrial project partner is the Corporate Research Sector of the Robert Bosch GmbH.

References

Alford, R.; Bercher, P.; and Aha, D. 2015a. Tight bounds for HTN planning. In *Proc. of the ICAPS*.

Alford, R.; Bercher, P.; and Aha, D. 2015b. Tight bounds for HTN planning with task insertion. In *Proc. of the IJCAI*.

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016a. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. of the ICAPS*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. of the IJCAI*.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proc. of the IJCAI*.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11(4):625–656.

Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2019a. Hierarchical planning in the IPC. In *Proc. of the Workshop on the IPC (WIPC)*.

Behnke, G.; Höller, D.; Bercher, P.; and Biundo, S. 2019b. More succinct grounding of HTN planning problems – Preliminary results. In *Proc. of the ICAPS Workshop on Hierarchical Planning*.

Behnke, G.; Höller, D.; and Biundo, S. 2018a. totSAT – Totally-ordered hierarchical planning through SAT. In *Proc. of the AAAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2018b. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proc. of the ICTAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proc. of the AAAI*.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proc. of the IJCAI*.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proc. of the IJCAI*.

Bercher, P.; Höller, D.; Behnke, G.; and Biundo, S. 2016. More than a name? On implications of preconditions and effects of compound HTN planning tasks. In *Proc. of the ECAI*.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. of the IJCAI*.

Bit-Monnot, A.; Smith, D. E.; and Do, M. 2016. Delete-free reachability analysis for temporal and hierarchical planning. In *Proc. of the ECAI*.

de Silva, L.; Lallement, R.; and Alami, R. 2015. The HATP hierarchical planner: Formalisation and an initial study of its usability and practicality. In *Proc. of the IROS*, 6465–6472.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. of AIPS*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the IJCAI*.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*.

González-Ferrer, A.; Fernández-Olivares, J.; and Castillo, L. 2009. JABBAH: A java application framework for the translation between business process models and HTN. In *Proc. of the Int. Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proc. of the ICAPS*.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On guiding search in HTN planning with classical planning heuristics. In *Proc. of the IJCAI*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

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

Schreiber, D.; Balyo, T.; Pellier, D.; and Fiorino, H. 2019. Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning. In *Proc. of the ICAPS*.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *Proc. of the AAAI*.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. of the AAMAS*.

Smith, D.; Frank, J.; and Cushing, W. 2008. The ANML language. In *Proc. of the Workshop on KEPS*.