# Hierarchical Task Networks Generated Using Invariant Graphs for IPC2020

**Damir Lotinac, Filippos Kominis, Anders Jonsson**[1]
[1] Universitat Pompeu Fabra, Roc Boronat 138, 08018 Barcelona, Spain.

**Abstract**

This paper describes HTN domains generated using PDDL description of a planning domain and a single representative instance. We also describe the algorithm used to generate the HTN domains. Two types of composite tasks that interact to achieve the goal of a planning instance are generated. One type of task achieves fluents by traversing invariants in which only one fluent can be true at a time. The other type of task applies a single action, which first involves ensuring that the precondition of the action holds. Finally we discuss differences between JSHOP2 and domains generated for IPC2020 in HDDL.

## Introduction

Hierarchical Task Networks models enable defining compound parameterized tasks which can reduce the search complexity if the adequate constraints can be identified during the modeling. The hierarchical structure enables the modeler to encode domain-specific knowledge. The expressiveness of HTNs can help to impose the constraints, which can in turn lead to a reduction in the search complexity. The hierarchy ideally imposes some constraints on how tasks can be decomposed. The more constrained the task network, the less search has to be performed in order to achieve a certain task.

HTN models are more expressive than STRIPS (Erol, Hendler, and Nau 1994), which along with the ability to construct parametric compound tasks allows for capturing domain-specific knowledge.

In this paper we describe the HTN domains and instances which were submitted to IPC2020 [1] and the algorithm which was used to generate them. We used PDDL instances from IPC-2000 and IPC-2002 as input.

To generate the HTN domains we use HTNPrec algorithm. The algorithm takes as input the PDDL description of a planning domain and a single representative instance. The approach is to generate HTNs that encode invariant graphs of planning domains. An invariant graph is similar to a lifted domain transition graph, but can be subdivided on types. To traverse an invariant graph we define two types of tasks: one that reaches a certain node of an invariant graph, achieving the associated fluent, and one that traverses a single edge of an invariant graph, applying the associated action. These two types of tasks are interleaved, in that the expansion of one type of task involves tasks of the other type.

We also describe differences between HTN domains generated by HTNPrec, HTNGoal and domains generated for the IPC2020. While HTNPrec and HTNGoal (Lotinac and Jonsson 2016) use a JSHOP2 (Nau et al. 2003) representation, for the IPC the domains are given in the HDDL format (Höller et al. 2020). Further some of the optimizations are not included in the IPC version. The HTN instances generated using JSHOP2 are solved with blind search, thus those HTN domains are meant to guide the search through the underlying invariant graph structures. In contrast HDDL domains are generated with minimal additions to the original PDDL domain.

## Hierarchical Task Networks

Our HTN definition is inspired by Geier and Bercher (2011). However, just as for STRIPS planning, we separate the definition into a domain part and an instance part. We also impose additional restrictions: a task network can contain at most one copy of each task, and task decomposition is limited to *progression*, always decomposing tasks with no predecessor.

An HTN domain is a tuple $\boldsymbol{h} = \langle P, A, C, M \rangle$ consisting of four sets of untyped function symbols. Specifically, $P$ is the set of *predicates*, $A$ is the set of *actions* (i.e. primitive tasks), $C$ is the set of *compound tasks* and $M$ is the set of *decomposition methods*. Predicates and actions are defined as for STRIPS domains but, unlike STRIPS domains, HTN domains are untyped and we allow negative preconditions.

Each method $m \in M$ has an associated tuple $\langle c, tn_m, \mathrm{pre}(m) \rangle$ where $c \in C$ is a compound task with the same arity as $m$, $tn_m$ is a *task network* and $\mathrm{pre}(m)$ is a set of preconditions, defined as for actions. The task network $tn_m = (T, \prec)$ consists of a set $T$ of pairs $(t, \varphi)$, where $t \in A \cup C$ is a task and $\varphi$ is an argument map from $m$ to $t$, and a partial order $\prec$ on the tasks in $T$.

Given an HTN domain $\boldsymbol{h}$, an HTN instance is a tuple

[1] The domains and the code are available at: https://github.com/dloti/pddl-to-htn

$\boldsymbol{s} = \langle \Omega, \text{init}, tn_I \rangle$, where $\Omega$ is a set of objects and init is an initial state. The instance $\boldsymbol{s}$ induces sets $P_\Omega$ and $A_\Omega$ of fluents and grounded actions, and sets $C_\Omega$ and $M_\Omega$ of grounded compound tasks and grounded methods, respectively. A grounded method $m[x] \in M_\Omega$ has associated tuple $\langle c[x], tn_m[x], \text{pre}(m[x]) \rangle$, where $c[x]$ is a grounded compound task and the precondition $\text{pre}(m[x])$ is derived as for grounded actions. The grounded task network $tn_m[x] = (T_x, \prec)$ is defined by $T_x = \{t[\varphi(x)] : (t, \varphi) \in T\}$. The initial grounded task network $tn_I = (\{t_I\}, \emptyset)$ contains a single grounded compound task $t_I \in C_\Omega$.

An HTN state $(s, tn)$ consists of a state $s \subseteq P_\Omega$ on fluents and a grounded task network $tn$. We use $(s, tn) \rightarrow_D (s', tn')$ to denote that an HTN state decomposes into another HTN state, where $tn = \langle T_x, \prec \rangle$ and $tn' = \langle T_y, \prec' \rangle$. A valid *progression decomposition* consists in choosing a grounded task $t \in T_x$ such that $t' \not\prec t$ for each $t' \in T_x$, and applying one of the following rules:

1. If $t$ is primitive, the decomposition is applicable if $\text{pre}(t) \subseteq s$, and the resulting HTN state is given by $s' = s \ltimes t, T_y = T_x \setminus \{t\}$ and $\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T_y\}$.
2. If $t$ is compound, a grounded method $m = \langle t, tn, \text{pre}(m) \rangle$ with $tn = (T_m, \prec_m)$ is applicable if $\text{pre}(m) \subseteq s$, and the resulting HTN state is given by $s' = s, T_y = T_x \setminus \{t\} \cup T_m$ and

$$\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T_y\} \cup \{(t', t_1) \in T_m \times T_y \mid (t, t_1) \in \prec\} \cup \prec_m .$$

The first rule removes a grounded primitive task $t$ from $tn$ and applies the effects of $t$ to the current state, while the second rule uses a grounded method $m$ to replace a grounded compound task $t$ with $tn_m$ while leaving the state unchanged. If there is a finite sequence of decompositions from $(s_1, tn_1)$ to $(s_n, tn_n)$ we write $(s_1, tn_1) \rightarrow_D^* (s_n, tn_n)$. An HTN instance $\boldsymbol{s}$ is solvable if and only if $(\text{init}, tn_I) \rightarrow_D^* (s_n, \langle \emptyset, \emptyset \rangle)$ for some state $s_n$, i.e. the initial HTN state $(\text{init}, tn_I)$ is decomposed into an empty task network. Let $\pi$ be the sequence of grounded actions extracted during such a decomposition; $\pi$ corresponds to a *plan* that results from solving $\boldsymbol{s}$.

## Invariants

In STRIPS planning, an exactly-1 invariant is a subset of fluents $F' \subseteq P_\Omega$ such that exactly one fluent in $F'$ is true at any moment. Formally, $|F' \cap \text{init}| = 1$ and any grounded action $a \in A_\Omega$ that adds a fluent in $F'$ deletes another. The Fast Downward planning system (Helmert 2009) uses the domain description of a STRIPS domain to detect lifted invariant candidates. Unlike Fast Downward, which grounds lifted invariants on actual instances, our algorithm operates directly on the lifted invariants.

In LOGISTICS, Fast Downward finds a single lifted invariant candidate {(`in ?o ?v`),(`at ?o ?p`)}, i.e. a set of predicates with associated arguments. In the given invariant, variable `?o` is *bound* while variables `?v` and `?p` are *free*. To ground the lifted invariant on an instance $\boldsymbol{p}$, we should create one mutex invariant $F'$ for each assignment of objects to the bound variables, obtaining each fluent in $F'$ by assigning objects to the free variables. In our running example, assigning the package `p1` to `?o` results in the following grounded mutex invariant:

{(`at p1 ap1`),(`at p1 ap2`),(`at p1 l1`),(`at p1 l2`),
(`in p1 t1`),(`in p1 t2`),(`in p1 a1`)}.

The meaning of the invariant is that across all LOGISTICS instances, a given object `?o` is either in a vehicle or at a location.

If a predicate $p \in P$ is not part of any invariant but there are actions that add and/or delete $p$, we create a new lifted invariant {($p$ `?o1` $\cdots$ `?ok`),($\neg p$ `?o1` $\cdots$ `?ok`)}. In this invariant, all variables `?o1`, ..., `?ok` are bound and an associated fluent can either be true or false.

Given a lifted invariant, our algorithm generates one or several invariant graphs. We do so by iterating over the actions of the domain and identifying which actions add and delete predicates in the same lifted invariant. When grounded, such actions have the effect of *changing* the fluent of an exactly-1 invariant that is currently true. An invariant graph is a representation of a lifted invariant in which the nodes are the predicates of the invariant and the edges are the actions used to change the predicate that is currently true. We use invariant graphs to infer which actions to perform in order to achieve a particular fluent of an exactly-1 invariant.

The reason why a given lifted invariant can generate multiple invariant graphs is that the *type* of the bound objects may be different for different actions. For example, in the LOGISTICS domain, all actions affect the lone invariant above. However, in the actions for loading or unloading a package, the bound object `?o` is a package, in the action for driving a truck `?o` is a truck, and in the action for flying an airplane `?o` is an airplane. Moreover, we can either load a package into a truck or an airplane. We use the actions to differentiate between types, possibly generating multiple invariant graphs for each lifted invariant.

To generate the invariant graphs induced by lifted invariants we go through each action, find each transition of each invariant that it induces (by pairing add and delete effects and testing whether the bound objects are identical), and map the types of the predicates to the invariant. We then either create a new invariant graph for the bound types or add nodes to an existing graph corresponding to the mapped predicate arguments.

Figure 1 shows the invariant graphs that we generate in LOGISTICS. In the top graph ($G_1$), the bound object is a package `?p`, in the middle graph ($G_2$) it is a truck `?t`, and in the bottom graph ($G_3$) it is an airplane `?a`. Note that the predicate `in` is not actually part of the two bottom graphs, since trucks and planes cannot be inside other vehicles. Nevertheless, the invariant still applies: a truck or plane can only be at a single place at once.

Each edge of an invariant graph corresponds to an action that deletes one predicate of the invariant and adds another. To do so, the arguments of the action have to include the arguments of both predicates, including the bound objects. In the figure, the invariant notation is extended to actions on

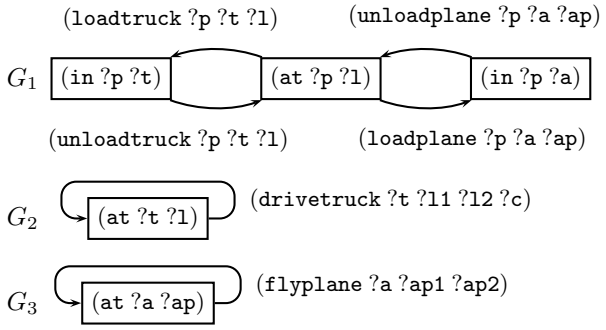


Figure 1: Invariant graphs $G_1$, $G_2$ and $G_3$ in LOGISTICS.

edges such that each argument of an action is either bound or free.

Even if actions preserve the invariant property, the initial state of a planning instance may violate the condition $|F' \cap \text{init}| = 1$, in which case $F'$ is not an exactly-1 invariant. To verify that a lifted invariant candidate corresponds to actual exactly-1 invariants, our algorithm needs access to the initial state of an example planning instance $\boldsymbol{p}$ of the domain. If this verification fails, the lifted invariant is not considered by the algorithm.

## Generating HTNs

In this section we describe the algorithm for generating the HTN domains. The idea is to construct a hierarchy of tasks that traverse the invariant graphs to achieve certain fluents. In doing so there are two types of interleaved tasks: one that achieves a fluent in a given invariant (which involves applying a series of actions to traverse the edges of the graph), and one that applies the action on a given edge (which involves achieving the preconditions of the action).

A planning domain is a tuple $\boldsymbol{d} = \langle \mathcal{T}, <, P, A \rangle$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a set of types, $<$ is an inheritance relation on types, $P$ is a set of typed function symbols called *predicates*, and $A$ is a set of typed function symbols called *actions*. Each action $a \in A$ has a set of preconditions $\text{pre}(a)$, a set of add effects $\text{add}(a)$ and a set of delete effects $\text{del}(a)$. Each element in these three sets is a pair $(p, \varphi)$ consisting of a predicate $p \in P$ and an argument map $\varphi$ from $a$ to $p$.

Given $\boldsymbol{d}$, a planning instance is a tuple $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, where $\Omega = \Omega_1 \cup \ldots \cup \Omega_n$ is a set of objects of each type.

Formally, our algorithm takes as input a STRIPS planning domain $\boldsymbol{d} = \langle \mathcal{T}, <, P, A \rangle$ and a planning instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$ and outputs an HTN domain $\boldsymbol{h} = \langle P, A', C, M \rangle$. The HTN domain $\boldsymbol{h}$ can then be used to solve any other instance of the domain. Specifically, for each instance $\boldsymbol{p}'$ of the planning domain $\boldsymbol{d}$, we construct an HTN instance $\boldsymbol{s}$. Solving the HTN induced by $\boldsymbol{d}$ and $\boldsymbol{s}$ returns a plan that can be adapted to solve $\boldsymbol{p}'$.

The input planning instance $\boldsymbol{p}$ is used for three purposes:

1. To verify that an invariant candidate is actually an invariant by testing the condition $|F' \cap \text{init}| = 1$.
2. To extract a subset of predicates $P_G \subseteq P$ that are part of the goal.
3. To perform goal ordering as described in a subsequent section.

The algorithm first constructs the invariant graphs $G_1, \ldots, G_k$ described above. In what follows we describe the components of the HTN domain $\boldsymbol{h}$.

The set $A'$ contains the following actions:

- Each action $a \in A$. For each element $\beta_k(a) \in \mathcal{T}$ of the type list of $a$, we add an additional precondition $(\beta_k(a), \varphi_k)$. where the argument map $\varphi_k$ maps the argument $x_k$ of $a$ to the lone argument of the type predicate $\beta_k(a)$, ensuring that argument $x_k$ has the correct type.

Note that only actions in $A$ add or delete predicates in the original set $P$. The set $C$ contains three types of compound tasks:

- For each predicate $p \in P$, a task achieve-$p$ with arity $\alpha(p)$.
- For each invariant graph $G_i$ and each $p \in P$ that is positive in $G_i$, a task achieve-$p$-$i$ with arity $\alpha(p)$.
- For each invariant graph $G_i$, each predicate $p$ in $G_i$, and each outgoing edge of $p$ (corresponding to an action $a \in A$), a task do-$p$-$a$-$i$ with arity $\alpha(a)$.

The task achieve-$p$ is a wrapper task that uses a task achieve-$p$-$i$ to achieve $p$ by traversing the edges of the invariant graph $G_i$. To traverse each edge of $G_i$, achieve-$p$-$i$ has to use a task of type do-$p$-$a$-$i$, which in turn uses tasks of type achieve-$p'$ to achieve the preconditions of $a$.

### Methods

The set $M$ contains the following decomposition methods. For simplicity, we use $x$ to denote an argument list, and define argument maps inline which are described in the text. We describe methods in pseudo-SHOP2 syntax in the following format:

(:method (⟨name⟩[⟨arguments⟩])
 (⟨precondition⟩)
 (⟨tasklist⟩))

For each method in the first line we specify a name and arguments, in the second line we give a precondition list, and finally in the third we specify the respective task list to which method decomposes. For clarity, we add an exclamation mark in front of primitive tasks.

- Methods for achieve-$p$

  The first type of compound task, achieve-$p$, has one associated method for each invariant graph $G_i$ in which $p$ appears. This method is defined as follows:

  (:method (achieve-$p[x]$)
   ($\neg p[x]$)
   (achieve-$p$-$i[x]$)).

  Intuitively this method delegates achieving $p$ to the task achieve-$p$-$i$ for some invariant graph $G_i$. The precondition $\neg p[x]$ ensures that $p$ is not currently true.

In addition, there is one method with empty task list which is applicable when $p$ already holds:

(:method (achieve-$p[x]$)
($p[x]$)
()).

- Methods for achieve-$p$-$i$

The second type of compound task, achieve-$p$-$i$, has one associated method for each predicate $q$ in the invariant graph $G_i$ and outgoing edge of $q$ (corresponding to an operator $o$):

(:method (achieve-$p$-$i[x]$)
($\neg p[x], q[\varphi_q(x)]$)
(do-$q$-$o$-$i[\varphi_o(x)]$, achieve-$p$-$i[x]$)).

Operator $o$ appears on an outgoing edge from $q$, i.e. $o$ deletes $q$. Intuitively, one way to achieve $p$ in $G_i$, given that we are currently at some different node $q$, is to traverse the edge associated with $o$ using the compound task do-$q$-$o$-$i$. After traversing the edge we recursively achieve $p$ from the resulting node. The argument map $\varphi_o$ should map the bound objects of $p$ to $o$ while leaving the remaining arguments of $o$ as free variables. The argument map $\varphi_q$ maps the bound objects of $p$ to $q$, and shares all free variables with $\varphi_o$ (since $q$ is a delete effect of $o$).

We also define a decomposition method for achieve-$p$-$i$ which is applicable when $p$ already holds and has empty task list:

(:method (achieve-$p$-$i[x]$)
($p[x]$)
())

- Method for do-$p$-$o$-$i$

The third type of compound task, do-$p$-$o$-$i$, has a single associated method. The aim is to apply operator $o$ to traverse an outgoing edge of $p$ in the invariant graph $G_i$. To do so, the task list has to ensure that all preconditions $p_1, \ldots, p_k$ of $o$ hold (excluding $p$, which has to hold to apply the method, as well as any static preconditions of $o$). We define the method as

(:method (do-$p$-$o$-$i[x]$)
($p[\varphi_p(x)]$)
(achieve-$p_1[\varphi_1(x)], \ldots,$ achieve-$p_k[\varphi_k(x)]$, $!o[x]$))

Here, the argument map $\varphi_j$, $1 \leq j \leq k$, maps the arguments of operator $o$ to the precondition $p_j$ of $o$. This mapping is given directly by the definition of operator $o$. Note that the decomposition achieves all preconditions of $o$ except $p$, then applies $o$.

When $p$ is the only precondition of operator $o$, task do-$p$-$o$-$i[x]$ is not needed since operator $o$ is always applicable as long as $p$ holds. In this case, whenever do-$p$-$o$-$i[x]$ appears in a decomposition method of a task achieve-$q$-$j$, we replace do-$p$-$o$-$i[x]$ directly with the operator $!o[x]$.

## Planning Instances

Once we have generated the HTN domain $\boldsymbol{h}$ we can apply it to any instance of the domain. Given a STRIPS instance $\boldsymbol{p} = \langle \Omega, \text{init}, \text{goal} \rangle$, we construct an HTN instance $\boldsymbol{s} = \langle \Omega, \text{init}', \langle$achieve-$p_1[x_1], \ldots,$ achieve-$p_k[x_k] \rangle \rangle$, given goal $= \{p_1[x_1], ..., p_k[x_k]\}$, as follows. The set of objects $\Omega = \Omega_1 \cup \cdots \cup \Omega_n$ is identical to that of $\boldsymbol{p}$. The initial state init$'$ is defined as init$' =$ init $\cup \{\tau_j[\omega] : \tau_j \in \mathcal{T}, \omega \in \Omega_j\} \cup \{$goal-$p[x] : p[x] \in$ goal$\}$. We thus mark the type $\tau_j$ of each object $\omega$ using the fluent $\tau_j[\omega]$, and we mark all fluents $p[x]$ in the goal state using the fluent goal-$p[x]$. The initial task network contains the achieve tasks which correspond to each fluent $p[x]$ in the goal state. The ordering of achieve tasks is imposed based on the order of goal fluents in the given PDDL instance.

# Optimizations

Achieving the preconditions of an action $a$ in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly. For this reason, we include the HTNPrec algorithm that uses a simple inference technique to compute a partial order in which to achieve the preconditions of $a$. We define a set of predicates whose value is supposed to persist, and check whether a path through an invariant graph is applicable given these persisting predicates. While doing so, only the values of bound variables are known, while free variables can take on any value. We match the bound variables of predicates and actions to determine whether an action allows a predicate to persist.

## Discussion

There are several differences between the HDDL domains and instances and JSHOP2 version generated by HTNPrec and HTNGoal. The HDDL version is closest to HTNPrec, since it does not apply goal order optimization. However, there are several differences to the HTNPrec algorithm as well. In this section we give tasks, methods and predicates which are not generated by the IPC2020 version of the algorithm.

Predicates which are not generated:

- visited-$p$, indicating that $p$ has already been visited during search.
- achieving-$p$, indicating that $p$ or another predicate in the same invariant are already being achieved.
- goal-$p$, indicating that a fluent derived from $p$ is a goal state.

Actions which are not generated:

- occupy-$i$, which marks each predicate in the invariant graph $G_i$ as being achieved.
- clear-$i$, which deletes visited-$p$ and achieving-$p$ for each predicate $p$ of the invariant graph $G_i$.
- test-$p$ with arity 0 and no effects, whose precondition tests if *all* goal fluents derived from $p$ hold.

The only task left out is solve whose decomposition achieves the goal condition JSHOP2 version. In HDDL the algorithm simply creates the task list by adding the achieve tasks in the order of appearance in the PDDL instance. This is under the assumption that the original instance can be solved under such restriction.

The predicates and actions mentioned above are used to guide the search over the HTN decompositions. As such these predicates are added having blind search in mind and while they should not hinder the perofmance of a heuristic planner, they can also be left out. HTNPrec and HTNGoal also have a different structure given that the resulting HTN consists of only one task to decompose. In contrast, for the IPC2020 we generated instances with task list consisting of achieve tasks.

## References

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, 1123–1128.

Geier, T., and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1955–1961.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. Hddl: An extension to pddl for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891.

Lotinac, D., and Jonsson, A. 2016. Constructing hierarchical task models using invariance analysis. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 1274–1282.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.