Constructing Hierarchical Task Models Using Invariance Analysis

Damir Lotinac and **Anders Jonsson** [1]

Abstract. Hierarchical Task Networks (HTNs) are a common model for encoding knowledge about planning domains in the form of task decompositions. We present a novel algorithm that uses invariant analysis to construct an HTN from the PDDL description of a planning domain and a single representative instance. The algorithm defines two types of composite tasks that interact to achieve the goal of a planning instance. 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. The resulting HTN can be applied to any instance of the planning domain, and is provably sound. We show that the performance of our algorithm is comparable to algorithms that learn HTNs from examples and use added knowledge.

Introduction

Hierarchical Task Networks, or HTNs, are a popular tool for encoding hierarchical structure into planning domains. In the past, HTNs have been successfully used in a variety of planning applications: military planning [26], Web service composition [32], unmanned air vehicle control [24], strategic game playing [30, 23], personalized patient care [29] and business process management [9], to name a few.

Although HTNs are at least as expressive as STRIPS planning [6], the main reason for their popularity is that they restrict actions choices. By excluding portions of the state space, search proceeds more quickly towards the goal, or in HTN terminology, towards generating a valid expansion of the initial task list. In the extreme case, each task has a single possible expansion, and planning is reduced to a simple traversal of the task hierarchy. Properly designing an HTN can be a time-consuming task for a human expert, but once this work is done, there is little need to optimize search.

Another powerful characteristic of HTNs is that tasks are parameterized, making it possible to encode knowledge about an entire planning domain, not just individual planning instances. Although identifying effective decomposition strategies for all instances of a domain can be arduous, once an HTN has been constructed for a planning domain, it can be used to solve an entire family of instances more efficiently.

Thus, the main reason that HTNs are frequently used in real-world applications is that they offer a potent mechanism for reducing the search effort required to solve a family of large-scale planning instances. This is also the reason that HTNs were so successful in the hand tailored track of early incarnations of the International Planning Competition (IPC): the participants were given access to the planning domains beforehand and designed HTNs that effectively narrowed the search to a tiny portion of the state space. It is not a coincidence that the HTN planner that achieved the largest coverage at IPC-2002 and was for a long time regarded as the state-of-the-art in HTN planning, SHOP2 [27], performs blind search in the task space to compute a valid expansion. Most of the work required to reduce the search effort is performed while designing the HTN, and once this work is done, there is little need to optimize search to solve each instance efficiently.

In summary, HTNs offer a mechanism for human experts to encode prior knowledge about a planning domain. Typically, this requires many hours of fine-tuning, debugging and testing. This is fine for specific planning applications in which the initial effort is compensated by the subsequent reduction in search time during successive applications of the planner. However, a large body of research in the planning community is dedicated to finding domain-independent approaches to planning. Traditionally, HTNs have not found a place in this body of research because of their domain-dependent emphasis.

A key motivation for this work is to explore whether it is possible to generate HTNs automatically in a domain-independent way. We also want to investigate whether such HTNs offer benefits similar to those designed by human experts. In the literature there exist two techniques that construct HTNs automatically [11, 35]. These techniques rely on annotated traces of plans that solve a set of instances from a domain.

In this paper we present a novel algorithm for generating HTNs automatically. Our algorithm takes as input the PDDL description of a planning domain and a single representative instance. Unlike previous approaches, the algorithm does not require solution plans for a subset of instances of the domain. Instead, our 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.

In experiments we test our approach on planning benchmarks from the International Planning Competition (IPC) and on the instances used in experiments with HTN-MAKER [11]. To solve the instances we use the SHOP2 planner [27], which performs blind search in the task space to compute a valid expansion. We show that the HTNs constructed by our algorithm solve all training and test set instances used to evaluate HTN-MAKER.

The rest of the paper is organized as follows. We first provide a

[1] Universitat Pompeu Fabra, Roc Boronat 138, 08018 Barcelona, Spain. Email: {damir.lotinac,anders.jonsson}@upf.edu.

background on planning and HTNs. We then introduce the concept of invariant graphs, and show how to generate invariant graphs for a given planning domain. We then describe our base algorithm for constructing HTNs. Next, we describe several optimizations on top of the base algorithm, and present experimental results. We conclude with a discussion of related work and possible directions for future work.

Planning

We consider the fragment of PDDL 2.1 [7] that models typed STRIPS planning domains with positive preconditions and goals. To represent planning domains we adapt a definition based on function symbols [1]. Given a set X, let X^n denote the set of vectors of length n whose elements are symbols in X. Given such a vector $x \in X^n$, let $x_k \in X$, $1 \leq k \leq n$, denote the k-th element of x.

We distinguish between typed and untyped function symbols. An untyped function symbol f has associated arity $\alpha(f)$. In addition, a typed function symbol f has an associated type list $\beta(f) \in \mathcal{T}^{\alpha(f)}$, where $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a set of types. Let F be a set of function symbols, and let $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$ be a set of objects, where Σ_i, $1 \leq i \leq n$, is a set of objects of type τ_i. We define $F_\Sigma = \{f[x] : f \in F, x \in \Sigma^{\alpha(f)}\}$ as the set of new objects obtained by applying each function symbol in F to each vector of objects in Σ of the appropriate arity. If f is typed, $f[x]$ has to satisfy the additional constraint $x_k \in \Sigma_{\beta_k(f)}$ for each k, $1 \leq k \leq \alpha(f)$, i.e. the type of each element in x is determined by the type list $\beta(f)$ of f.

Let f and g be two function symbols in F. An *argument map from f to g* is a function $\varphi : \Sigma^{\alpha(f)} \to \Sigma^{\alpha(g)}$ mapping arguments of f to arguments of g. An argument map φ allows us to map each object $f[x] \in F_\Sigma$ to an object $g[\varphi(x)] \in F_\Sigma$. In PDDL, argument maps have a restricted form: each element in $\varphi(x)$ is either an element from x or a constant object in Σ independent of x. WLOG we assume that argument maps are well-defined for typed function symbols.

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 $\mathrm{pre}(a)$, a set of add effects $\mathrm{add}(a)$ and a set of delete effects $\mathrm{del}(a)$. Each element in these three sets is a pair (p, φ) consisting of a predicate $p \in P$ and an argument map φ from a to p.

Given $\boldsymbol{d}$, a planning instance is a tuple $\boldsymbol{p} = \langle \Omega, \mathrm{init}, \mathrm{goal} \rangle$, where $\Omega = \Omega_1 \cup \ldots \cup \Omega_n$ is a set of objects of each type. The instance $\boldsymbol{p}$ implicitly defines a set of fluents P_Ω and a set of grounded actions A_Ω. A grounded action $a[x] \in A_\Omega$ has a set of preconditions $\mathrm{pre}(a[x])$, a set of add effects $\mathrm{add}(a[x])$ and a set of delete effects $\mathrm{del}(a[x])$. Each element in these sets is a fluent $p[\varphi(x)] \in P_\Omega$, where (p, φ) is the associated precondition or effect of the action a. The initial state $\mathrm{init} \in P_\Omega$ and goal state $\mathrm{goal} \in P_\Omega$ are both subsets of fluents. We often abuse notation by dropping the argument x of elements in P_Ω and A_Ω.

A state $s \subseteq P_\Omega$ is a subset of fluents that are true, while fluents in $P_\Omega \setminus s$ are false. A grounded action $a \in A_\Omega$ is applicable in s if and only if $\mathrm{pre}(a) \subseteq s$, and the result of applying a in s is a new state $s \ltimes a = (s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)$. A plan for $\boldsymbol{p}$ is a sequence of grounded actions $\pi = \langle a_1, \ldots, a_m \rangle$ such that a_j, $1 \leq j \leq m$, is applicable in $\mathrm{init} \ltimes a_1 \ltimes \cdots \ltimes a_{j-1}$, and π solves $\boldsymbol{p}$ if it reaches the goal state, i.e. if $\mathrm{goal} \subseteq \mathrm{init} \ltimes a_1 \ltimes \cdots \ltimes a_m$.

We use the LOGISTICS domain to illustrate the PDDL definition of a planning domain and instance. The types and predicates of LOGISTICS are given by

```
(:types truck airplane − vehicle
        package vehicle − physobj
        airport location − place
        city place physobj − object)

(:predicates (incity ?loc − place ?city − city)
             (at ?obj − physobj ?loc − place)
             (in ?pkg − package ?veh − vehicle)).
```

An example action for LOGISTICS is given by

```
(:action loadtruck
 :parameters   (?p − pkg ?t − truck ?l − place)
 :precondition (and (at ?t ?l) (at ?p ?l))
 :effect       (and (not (at ?p ?l)) (in ?p ?t))).
```

We use the following LOGISTICS instance as a running example:

```
(:objects a1 − airplane ap1 ap2 − airport
          c1 c2 − city l1 l2 − location
          t1 t2 − truck p1 − package)

(:init (at p1 l1) (at a1 ap2)
       (at t1 l1) (incity l1 c1) (incity ap1 c1)
       (at t2 l2) (incity l2 c2) (incity ap2 c2))

(:goal (and (at p1 ap1)))).
```

Hierarchical Task Networks

Our HTN definition is inspired by Geier and Bercher [8]. 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, φ), where $t \in A \cup C$ is a task and φ 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 $\boldsymbol{s} = \langle \Omega, \mathrm{init}, tn_I \rangle$, where Ω is a set of objects and init is an initial state. The instance $\boldsymbol{s}$ induces sets P_Ω and A_Ω of fluents and grounded actions, and sets C_Ω and M_Ω of grounded compound tasks and grounded methods, respectively. A grounded method $m[x] \in M_\Omega$ has associated tuple $\langle c[x], tn_m[x], \mathrm{pre}(m[x]) \rangle$, where $c[x]$ is a grounded compound task and the precondition $\mathrm{pre}(m[x])$ 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 $(s, tn) \to_D (s', tn')$ 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 ***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 (init, tn_I) is decomposed into an empty task network. Let π be the sequence of grounded actions extracted during such a decomposition; π corresponds to a *plan* that results from solving ***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 [10] 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 ***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.

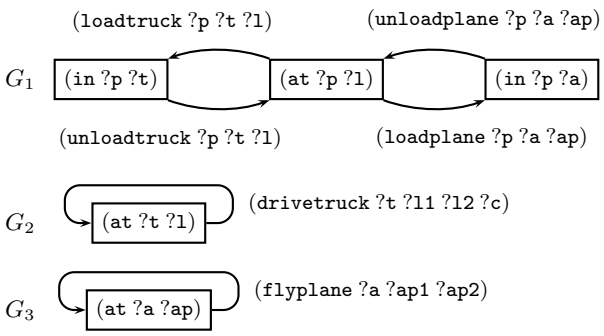

Figure 1. Invariant graphs G_1, G_2 and G_3 in LOGISTICS.

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 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 ***p*** of the domain. If this verification fails, the lifted invariant is not considered by the algorithm.

Generating HTNs

In this section we describe our algorithm for automatically generating 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).

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 $P' \supseteq P$ extends P with the following predicates:

- For each predicate $p \in P$, a predicate `visited-`p with arity $\alpha(p)$ indicating that p has already been visited during search.
- For each predicate $p \in P$, a predicate `achieving-`p with arity $\alpha(p)$ indicating that p or another predicate in the same invariant are already being achieved.
- For each goal predicate $p \in P_G$, a predicate `goal-`p with arity $\alpha(p)$ indicating that a fluent derived from p is a goal state.
- For each type $\tau \in \mathcal{T}$, a type predicate τ with arity 1.

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 φ_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.
- For each $p \in P$, an action `visit-`p with arity $\alpha(p)$ that marks p as visited by adding `visited-`p.
- For each invariant graph G_i, an action `occupy-`i whose arity equals the number of bound objects in G_i, and that marks each predicate in G_i as being achieved.
- For each invariant graph G_i, an action `clear-`i whose arity equals the number of bound objects in G_i, and that deletes `visited-`p and `achieving-`p for each predicate p of G_i.
- For each goal predicate $p \in P_G$, an action `test-`p with arity 0 and no effects, whose precondition tests if *all* goal fluents derived from p hold.

Note that only actions in A add or delete predicates in the original set P. The set C contains four 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 $\alpha(a)$.
- A task `solve` with arity `0`.

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. The task `solve` serves as the root task of the HTN and recursively achieves the goal by applying one task of type `achieve-`p at a time.

Finally, the set M contains the following decomposition methods. 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 name and arguments, in the second line we give precondition list, and finally in the third we specify task list to which method decomposes. For clarity, we add an exclamation mark in front of primitive tasks. The first type of compound task, `achieve-`p, has one associated method for each invariant graph G_i in which p appears. An outline of this method is given by

(:method (`achieve-`$p[x]$)
 ((¬`achieving-`$p[x]$))
 ((!`occupy-`$i[\varphi_i(x)]$) (`achieve-`p-$i[x]$) (!`clear-`$i[\varphi_i(x)]$))).

The argument map φ_i maps the arguments of p to the bound variables of the invariant graph G_i. Intuitively this method delegates achieving p to the task `achieve-`p-i for some invariant graph G_i. The method first adds `achieving-`p' for each predicate p' in G_i, and clears the flags after achieving p. The precondition (¬`achieving-`$p[x]$) prevents us from achieving p if it is part of an occupied invariant graph, which could potentially lead to an infinite recursion.

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

(:method (`achieve-`p-$i[x]$)
 ($(p'[\varphi'(x)])$ (¬`visited-`$p'[\varphi'(x)]$))
 ((!`visit-`$p'[\varphi'(x)]$) (`do-`p'-a-$i[\varphi_a(x)]$) (`achieve-`p-$i[x]$))).

Action a appears on an outgoing edge from p', i.e. a deletes p'. Intuitively, one way to achieve p in G_i, given that we are currently at some different node p', is to traverse the edge associated with a using the compound task `do-`p'-a-i. Before doing so we mark p' as visited to prevent us from visiting p' again. After traversing the edge we recursively achieve p from the resulting node. The argument map φ' should set the bound objects of p' while leaving other arguments of p' as free variables. Likewise, the argument map φ_a should set the bound objects of a. The precondition (¬`visited-`$p'[\varphi'(x)]$) prevents us from visiting the same node p' twice. In essence, the result is a depth-first search through the invariant graph G_i, which stops when we reach p. Recall that the flags `visited-`p and `achiveing-`p are cleared by the parent method once we reach p.

To stop the recursion we define a "base case" method:

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

This method is applicable when p already holds and has empty task list.

The third type of compound task, `do-`p-a-i, has only one associated method. The aim is to apply action a 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 a hold (excluding p, which holds by definition, as well as any static preconditions of a). We define the method as

(:method (do-p-a-$i[x]$)
 (($p[\varphi(x)]$))
 ((($\texttt{achieve-}p_1[\varphi_1(x)]$) $\cdots$ ($\texttt{achieve-}p_k[\varphi_k(x)]$)) (!$a[x]$))

Here, (p, φ) is the precondition of a associated with p, while (p_i, φ_i), $1 \leq i \leq k$, are the remaining preconditions of a. The decomposition achieves all preconditions of a, then applies a. Note that if action a has no preconditions except p, the mutual recursion stops since the decomposition does not contain any task of type achieve-p_i. In this case our approach is to simplify the definition of other methods by replacing any instance of do-p-a-i with the action a itself.

The fourth type of compound task, solve, has one method for each predicate $p \in P_G$ that appears in the goal state:

(:method (solve)
 ((goal-$p[x]$)($\neg(p[x])$))
 ((achieve-$p[x]$)(solve))

Here, x are free parameters. Whenever a predicate in the goal state does not hold, we achieve it and recursively call solve. Again, we need a base case method to stop the recursion:

(:method (solve)
 ()
 ((!test-p_1) $\cdots$ (!test-p_k))

Here, $p_1, \ldots, p_k$ are the predicates in P_G, i.e. those that appear in the goal state. Since each action test-p_i, $1 \leq i \leq k$, checks whether all goal fluents associated with p_i hold, this method is only applicable when the goal state holds. The reason this check is not made in the precondition of the method itself is that SHOP2 only supports forall clauses in action preconditions, not method preconditions.

To restrict the choices made when traversing the HTN, we impose a total order on all task lists of methods, except the tasks for achieving the preconditions of a in do-p-a-i. The reason is that it may be difficult to determine in which order to achieve the preconditions of an action.

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 $\boldsymbol{s} = \langle \Omega, \text{init}', \langle \texttt{solve}, \emptyset \rangle \rangle$ 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

$$\text{init}' = \text{init} \cup \{\tau_j[\omega] : \tau_j \in \mathcal{T}, \omega \in \Omega_j\} \cup \{\texttt{goal-}p[x] : p[x] \in \text{goal}\}.$$

We thus mark the type τ_j of each object ω 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 single grounded compound task solve.

We show that the HTN translation is sound. The translation is only complete if we allow goals and preconditions to be achieved in any order; we violate this condition below.

Theorem 1 *Let π be a plan that results from a valid decomposition for $\boldsymbol{s}$, and construct π' by removing grounded actions of type* visit-p, test-p, occupy-i *and* clear-i. *Then π' solves $\boldsymbol{p}$.*

Proof sketch Recall that an HTN state is given by (s, tn) for a planning state s and task network tn. When we decompose an HTN state, s is only changed if we use primitive tasks, i.e. changes to s are caused precisely by the elements of π. Consider the fluent set P_Ω induced by $\boldsymbol{p}$. The grounded actions of the types removed do not add or delete fluents in P_Ω. Conversely, the grounded actions that remain in π' are those of the original action set A_Ω, which *only* add or delete fluents in P_Ω.

Let $\pi' = \langle a_1, \ldots, a_m \rangle$ be the sequence of grounded actions from A_Ω. We can only decompose an HTN state (s, tn) using a primitive task a_j, $1 \leq j \leq m$, if the precondition of a_j holds in s. It follows that $\text{pre}(a_j) \subseteq \text{init} \ltimes a_1 \ltimes \cdots \ltimes a_{j-1}$ for each j, $1 \leq j \leq m$. Hence π' is a plan for $\boldsymbol{p}$.

To prove that the plan π' solves $\boldsymbol{p}$, it remains to show that the goal state goal of $\boldsymbol{p}$ holds after applying π' in init. Since solve is recursively applied during HTN decomposition, the sequence π has to have suffix $\langle \texttt{test-}p_1, \ldots, \texttt{test-}p_k \rangle$, the decomposition of the only base case method for solve. This action sequence has no effects and is only applicable if goal holds. Hence goal has to hold after the last action in π' since the removed actions have no effects on fluents in P_Ω.

Example

In LOGISTICS, our algorithm generates two wrapper tasks achieve-in and achieve-at, and four tasks achieve-in-1, achieve-at-1, achieve-at-2, and achieve-at-3, corresponding to the predicates in the three invariant graphs. The task achieve-at-1 has five associated methods: one for each edge of the graph G_1, plus the base case method.

The algorithm also generates six tasks do-at-loadtruck-1, do-at-loadplane-1, do-at-unloadtruck-1, do-at-unloadplane-1, do-at-drivetruck-2, and do-at-flyplane-3, corresponding to the six edges of the graphs. The latter two do not have preconditions besides at (the predicate incity in the precondition of drivetruck is static). The remaining four tasks each achieve a single precondition: the truck or plane being at the associated place.

To illustrate the tasks and associated methods we sketch the task expansions of the HTN instance generated from our running example. The only goal is (at p1 ap1), so the task solve has a single valid decomposition that contains the task (achieve-at p1 ap1). Table 1 shows the first five task expansions of the HTN instance. In each case, the compound task to be decomposed is underlined, and the new tasks inserted as a result of the decomposition are colored in the next step.

The second decomposition is produced by the lone method for (achieve-at p1 ap1). The current node associated with p1 in G_1 is (at p1 l1), with two outgoing edges, corresponding to actions loadtruck and loadplane. Applying the method for (achieve-at-1 p1 ap1) associated with (at p1 l1) and loadtruck produces the third expansion. The only method for (do-at-loadtruck-1 p1 t1 l1) expands to (achieve-at t1 l1), which in turn expands to (achieve-at-2 t1 l1) (the last expansion shown).

Optimizations

In this section we discuss several optimizations of the base algorithm for generating HTNs.

Ordering Preconditions

Achieving the preconditions of an action a in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly to find a correct order. For this reason, we include an extension of our algorithm that uses a simple inference technique to compute a partial order in which to achieve the preconditions of a.

(solve)	(achieve-at p1 ap1)	(!occupy-1 p1)	(!occupy-1 p1)	(!occupy-1 p1)
(solve)		(achieve-at-1 p1 ap1)	(!visit-at p1 l1)	(!visit-at p1 l1)
		(clear-1 p1)	(do-at-loadtruck-1 p1 t1 l1)	(!occupy-2 t1)
		(solve)	(achieve-at-1 p1 ap1)	(achieve-at-2 t1 l1)
			(!clear-1 p1)	(!clear-2 t1)
			(solve)	(!loadtruck p1 t1 l1)
				(achieve-at-1 p1 ap1)
				(!clear-1 p1)
				(solve)

Table 1. The first five task expansions of the HTN instance generated from the running example in LOGISTICS. The colored tasks are those added by the decomposition of the underlined task in the previous step.

```
1:  function ORDER(a, p)
2:      V ← pre(a) \ {p}, Z ← ⟨⟩
3:      repeat
4:          for p' ∈ V do
5:              W ← {p} ∪ V \ {p'}
6:              for each invariant graph G_j containing p' do
7:                  Perform a backwards BFS in G_j from p'
8:                  Test if paths applicable when W persists
9:              end for
10:             if each path achieving p' is applicable then
11:                 V ← V \ {p'}
12:                 Z ← ⟨p', Z⟩
13:             end if
14:         end for
15:     until V, Z converge
16:     return (V, Z)
17: end function
```

Figure 2. Algorithm ordering all preconditions of a except p.

We define a subset 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 the bound variables are known, while free variables can take on any value. Matching the bound variables of predicates and actions enables us to determine whether an action allows a predicate to persist.

Consider a task of type do-p-a-i, i.e. the purpose of the task is to apply action a in order to delete p. Figure 2 shows how to order all preconditions of action a except p. In the algorithm, V is the set of preconditions to be ordered, while Z is a sequence of preconditions, initially empty. The algorithm considers one precondition $p' \in V$ at a time and checks if it is possible to achieve p' while all remaining preconditions persist. If so, we remove p' from V and place it first in Z. We then iterate until no more preconditions can be removed from V, and return (V, Z). In the method m associated with do-p-a-i, the preconditions in Z can be achieved in order. On the other hand, we cannot say anything about the order of preconditions that remain in V.

Goal Ordering

Just as for preconditions, achieving goals in any order results in significant backtracking. To order the goals we implement an algorithm similar to the one for ordering preconditions. While the ordered preconditions are coded into the HTN, the goals are different for each instance of the domain. Since HTNs are instance-independent, our approach is to define new tasks that compute a goal ordering as a preprocessing step.

To accomplish this, we first order the goals of the representative planning instance $\boldsymbol{p}$ passed as input to the algorithm. We run the precondition ordering algorithm on the set of goal predicates $P_G \subseteq P$, i.e. predicates whose associated fluents appear in the goal. Given an ordering of the predicates in P_G, we then order the set of fluents of each predicate $p \in P_G$ using a similar algorithm. To do so, the invariant graphs need to be partially grounded on each pair of fluents to be ordered.

For each $p \in P_G$ and each pair of fluents $p[x]$ and $p[y]$, we check if $p[y]$ is achievable when $p[x]$ persists (i.e. we are not allowed to delete $p[x]$). Each invariant graph that contains p is partially grounded on $p[x]$, while the preconditions of actions that directly achieve p are grounded on $p[y]$. If this grounding violates the invariant, $p[y]$ should be ordered before $p[x]$. Once the invariant is invalidated by partial grounding, the algorithm stores the indices of the parameters of p that invalidated the invariant.

As an example, in the BLOCKS domain, $P_G = \{\texttt{on}\}$, so the method for `solve` always decomposes to `achieve-on`. Figure 3 shows one of the invariant graphs in BLOCKS that contains the predicate `on`. We test each pair of goal fluents to establish an order among them. Consider two goal fluents (`on a b`) and (`on b c`) that both refer to block `b`. If we fix (`on a b`) and attempt to achieve (`on b c`), the only operator (`stack b c`) that directly achieves (`on b c`) has precondition (`holding b`), which violates the invariant since the fluent (`on a b`) should persist. Thus fluent (`on b c`) should be ordered before fluent (`on a b`). We can generalize this knowledge and derive a rule that whenever two goal fluents of type `on` have the same object as the first and second parameter, respectively, the former should be ordered before the latter.

To implement the goal ordering mechanism we introduce a new compound task `order` with arity 1 whose associated method uses the ordering rule from above to order the goal fluents of a single predicate $p \in P_G$. The argument of `order` is a new object that encodes a *counter* that starts at 0, and the result of decomposing `order` is assigning a count to each fluent in the goal. We force `order` to be the first task expanded. We then add an argument to the `solve` task that represents a counter, and change the initial task network to contain (`solve` 0), i.e. the initial count is 0. The recursive method for `solve` requires us to always achieve the goal fluent corresponding to the current count, and we are only allowed to increment the count whenever the current goal fluent already holds. To ensure that all goal fluents are eventually achieved we reset the counter to 0 each time we achieve a goal fluent.

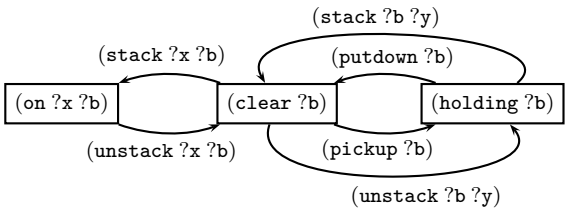

Figure 3. Invariant graph in the BLOCKS domain.

Sorting the Bindings of Free Variables

Apart from deciding which method to use to decompose a compound task and in which order to decompose partially ordered tasks in a task network (whenever the task network does not impose a total order on tasks), another non-deterministic choice made by SHOP2 is how to assign objects to the free variables of a method (e.g. the argument x of the predicate p in the recursive method for decomposing the `solve` task). Since SHOP2 performs blind search, it backtracks over all possible bindings to these free variables.

In the recursive method for decomposing the task `achieve-`p`-`i, the free variables determine which action to perform from the current node of the corresponding invariant graph. If we do not impose any order on variable bindings, SHOP2 iterates over such actions in an arbitrary fashion. As a consequence, the decompositions of `do-`p`-`a`-`i tasks do not immediately aim for the target node, even if this node is just one step away. We exploit the ability of SHOP2 to order the bindings of free variables by imposing an order that first attempts to traverse an edge to the target node. If this is not possible, blind search later explores all other possibilities.

Results

We ran our algorithm on all instances in the training and test set of HTN-MAKER (LOGISTICS, SATELLITE and BLOCKS). We also tested the algorithm in other STRIPS planning domains from the IPC-2000 and IPC-2002. The experiments were performed with two versions of our algorithm. The base algorithm that achieves the preconditions and goals in any order was slow in testing, so we activated precondition ordering in both versions. The first version, HTNPrec, achieves the goals in the order they appear in the PDDL definition. The second version, HTNGoal, implements our goal ordering and free variable sorting strategies in addition to precondition ordering. In all experiments we used JSHOP (the Java implementation of SHOP2), which uses blind search to compute a valid expansion. We used a memory limit of 4GB and a timeout of 1,800 seconds. We compare to the latest results of HTN-MAKER (WeakS), which uses a C++ implementation of SHOP2 and one hour of CPU time to solve the instances [12]. However, the instances from that paper are not publicly available, so we could not compare directly, and we just show their reported coverage results for reference.

Table 2 shows the coverage results on instances from HTN-MAKER's experiments. We tested HTN-MAKER's output domains from the fifth and final trial [11] and we ran it over the test set only, while we ran our algorithm over both the test and training sets of the experiment instances. To achieve the reported results, HTN-MAKER needed 420 training instances in BLOCKS and 75 training instances in LOGISTICS and SATELLITE, while our algorithm used the domain and a single example instance with no need to solve the instance or annotating a plan beforehand. For ROVERS and ZENOTRAVEL we report coverage over the IPC-2002 instances for HTNPrec and HTNGoal. The improvement of HTNGoal over HTNPrec in ZENOTRAVEL is due to free variable sorting (since airplanes can fly directly to the target destination). HTNGoal performs better in other domains as well; however, in terms of coverage of HTN-MAKER's instances there is little difference.

	HTNPrec	HTNGoal	HTN-MAKER	WeakS
LOGISTICS	100 %	100%	100%	93,6%
SATELLITE	100 %	100%	92%	100%
BLOCKS	100 %	100%	63,5%	99%
ROVERS	100 % *	100% *	-	100%
ZENOTRAVEL	20% *	100% *	-	99,8%

Table 2. Coverage of instances from HTN-MAKER's experiments. Scores marked with an asterisk (*) are taken from IPC instances.

The instances used in HTN-MAKER's experiments are generated by a random generator with very few objects (e.g. a maximum of five blocks in BlocksWorld), while our HTNs can be used to solve much larger instances. Therefore we ran both the HTNPrec and HTNGoal versions of our algorithm on other benchmark instances from IPC-2000 and IPC-2002. The results from these experiments appear in Table 3. For example, in BLOCKS, IPC benchmark instances include up to 50 blocks. On those benchmark instances HTNGoal achieves full coverage of BLOCKS, SATELLITE, ROVERS, LOGISTICS, ZENOTRAVEL and MICONIC. The improvement in average number of backtracks is most clearly visible in the BLOCKS domain, and is due to our goal ordering strategy. It also allowed for an increase in number of instances solved in the DEPOTS domain, as these two domains are the most sensitive to goal ordering. The combination of goal ordering and free variable sorting shows an increase in performance in other domains as well. This allows HTNGoal to solve all instances in many domains, with very few backtracks. These domains however tend to be the ones with a lower branching factor in the invariant graphs.

It is important to note that unsolved instances are not due to failed decompositions. Rather, the alloted time was insufficient to complete the search. While the results of our approach are comparable to those of HTN-MAKER, in some domains the generated HTNs do not perform well due to excessive backtracking (e.g. we do not solve any instances of the IPC-2002 FREECELL domain, which is more puzzle-like and therefore harder to serialize as our HTNs do by achieving one goal fluent at a time). On the other hand in DRIVERLOG, the algorithm does not solve many instances due to our goal ordering strategy. In this domain, using the lifted invariant graphs finds only goal predicate orderings, which is insufficient, and thus achieved goals often have to be unachieved by the `solve` task. Apart from that, in this domain, a system of "link-paths" is used for both drivers and trucks, which means that a path needs to be found (if one exists) for each location that happens to be a goal or subgoal location.

Related Work

Our approach to generating HTNs from a single planning instance and using them to solve larger instances of the same planning domain can be viewed as a form of generalized planning, which has received

		HTNPrec			HTNGoal		
Domain	Instances	#s	t	#b	#s	t	#b
Freecell	60	0	-	-	0	-	-
Blocks	103	24	91	8118	103	0.6	0.4
Rovers	20	20	0.6	1.2	20	0.5	1.1
Logistics	80	80	2.9	44	80	2.4	23.7
Driverlog	20	0	-	-	3	0.4	1.3
Zenotravel	20	4	25.6	4101	20	0.5	0.3
Miconic	150	150	0.66	0	150	0.63	0
Satellite	20	7	0.59	1.2	20	0.37	0.04
Depots	22	8	22.6	1867	17	88.4	8108

Table 3. Results in the IPC-2000 and IPC-2002 domains, with the total number of instances of each domain shown in brackets. For each solver we report number of solved instances (#s), average time in seconds (t) and average number of backtracks in thousands (#b) respectively

a lot of recent attention, most notably in the form of the learning track of the IPC. One popular approach to generalized planning is to identify macros [2, 20, 25, 28], i.e. sequences of operators that frequently appear in the solutions to example instances. Once identified, such macros can then be inserted into the action space of larger instances in order to speed up search. However, even though macros can be parameterized, they do not offer the same flexibility as HTNs in terms of representing a solution to all instances of a planning domain.

Another approach to generalized planning is to learn reactive policies for planning domains [15, 17, 22, 33]. A third approach is to learn domain-specific knowledge in order to improve heuristic estimates computed during search [4, 34]. In contrast to most of these techniques, which are *inductive*, ours is a *generative* approach, as we construct HTNs directly by analyzing the domain model.

Achieving fluents by traversing the edges of domain transition graphs is the strategy used by DTGPlan [3] and similar algorithms. There also exist other inference techniques that can solve many individual instances backtrack-free [18, 19]. The novelty of our approach compared to previous work is the ability to do this in an instance-independent way.

The most popular approaches for generating hierarchical task models are learning from examples. However these approaches not only need to learn from examples but also rely on some given task-subtask decompositions [14], annotated plans [11], or added concepts [16]. In contrast, our algorithm requires less semantic information, and although we could use the representative instance (or multiple instances) to generate solution plans for learning, these plans would still have to be annotated to be used by HTN-MAKER.

A basic compilation from STRIPS to HTN was defined by Erol et al [6]. This compilation constructs primitive tasks for each STRIPS operator and a single compound task. However this compilation is used only for theoretical purposes, as it does not impose more restrictions on the task network. Our work is also related to other approaches to hierarchical planning [13, 21, 5], with the difference that we generate the hierarchies automatically.

Conclusion

In this paper we present a domain-independent algorithm for generating HTNs. All the algorithm needs is a PDDL description of the planning domain and a single representative instance, which is needed for three reasons: 1) to validate invariant candidates; 2) to determine which predicates appear in the goal state; and 3) to establish a goal ordering. While other approaches learn from solved instances, ours can be viewed as a form of compilation. We show that our algorithm is competitive with state-of-the-art algorithms for automatically learning HTNs from solved instances. The algorithm is not complete if we do not allow for preconditions and goals to be achieved in any order, which is the case in our experimental setting.

While the results of our approach are comparable to those of HTN-MAKER, in some domains due to the branching factor of the invariant graphs the generated HTNs do not perform well. In FREECELL, apart from a high branching factor, the domain used in the competition is encoded so that it uses auxiliary predicates like `number` and `successor`. This causes a high arity of the actions (implying that there are more possible bindings of objects to the argument of actions), and is used to impose an order for many different stacks. It is possible that our approach would solve at least a few instances if a different representation were used.

Although the success of the algorithm is limited in some domains, we believe that there are still many potential benefits. The algorithm takes a fraction of a second to generate HTNs given a PDDL domain and a single example instance. The example instance does not need to be solved, and no plan traces are required. Since the algorithm is domain-independent it does not require any intervention and is easy to run. Therefore the resulting HTN could potentially be useful even in cases where it does not perform well right after the compilation, e.g. by extracting useful subtasks.

The avenue for future research that we find most promising is to test different restrictions on the invariant graphs. If the representative instance can still be solved under some restriction, the resulting HTN may still be able to solve other instances, and the restriction has the effect of reducing the branching factor. In essence, this mechanism would reduce the number of ways to traverse the invariant graphs. Another possible extension is to identify and prune methods that are not needed to solve the HTN instances.

User specified heuristics for HTNs have shown useful for automatically generated HTNs [31]. For example such heuristic would easily resolve path-finding problems in domains like DRIVERLOG. Therefore another option is to construct heuristics, which would be used to guide task-subtask decompositions and sort the bindings of free variables in order to direct search more efficiently.

Acknowledgment

This work is partially supported by the MINECO/FEDER grant TIN2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain.

REFERENCES

[1] C. Bäckström, A. Jonsson, and P. Jonsson, 'Automaton Plans', *Journal of Artificial Intelligence Research*, **51**, 255–291, (2014).

[2] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer, 'Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators', *Journal of Artificial Intelligence Research*, **24**, 581–621, (2005).

[3] Y. Chen, R. Huang, and W. Zhang, 'Fast Planning by Search in Domain Transition Graphs', in *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pp. 886–891, (2008).

[4] T. de la Rosa, S. Jiménez, R. Fuentetaja, and D. Borrajo, 'Scaling up Heuristic Planning with Relational Decision Trees', *Journal of Artificial Intelligence Research*, **40**, 767–813, (2011).

[5] M. Elkawkagy, P. Bercher, B. Schattenberg, and S. Biundo, 'Improving Hierarchical Planning Performance by the Use of Landmarks', in *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI'12)*, (2012).

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

[7] Maria Fox and Derek Long, 'Pddl2. 1: An extension to pddl for expressing temporal planning domains.', *J. Artif. Intell. Res.(JAIR)*, **20**, 61–124, (2003).

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

[9] A. González-Ferrer, J. Fernández-Olivares, and L. Castillo, 'From Business Process Models to Hierarchical Task Network Planning Domains', *Knowledge Engineering Review*, **28**(2), 175–193, (2013).

[10] M. Helmert, 'Concise finite-domain representations for PDDL planning tasks', *Artificial Intelligence*, **173**, 503–535, (2009).

[11] C. Hogg, H. Munoz-Avila, and U. Kuter, 'HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required', in *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pp. 950–956, (2008).

[12] Chad Hogg, Héctor Muñoz-Avila, and Ugur Kuter, 'Learning hierarchical task models from input traces', *Computational Intelligence*, (2014).

[13] R. Holte, M. Perez, R. Zimmer, and A. MacDonald, 'Hierarchical A*: Searching Abstraction Hierarchies Efficiently', in *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, pp. 530–535, (1996).

[14] Okhtay Ilghami, Dana S Nau, Héctor Munoz-Avila, and David W Aha, 'Camel: Learning method preconditions for htn planning', in *AIPS*, pp. 131–142, (2002).

[15] R. Khardon, 'Learning Action Strategies for Planning Domains', *Artificial Intelligence*, **113**(1-2), 125–148, (1999).

[16] Pat Langley and Dongkyu Choi, 'Learning recursive control programs from problem solving', *The Journal of Machine Learning Research*, **7**, 493–518, (2006).

[17] J. Levine and D. Humphreys, 'Learning Action Strategies for Planning Domains Using Genetic Programming', in *EvoWorkshops*, volume 2611 of *Lecture Notes in Computer Science*, pp. 684–695, (2003).

[18] N. Lipovetzky and H. Geffner, 'Inference and Decomposition in Planning Using Causal Consistent Chains', in *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, (2009).

[19] N. Lipovetzky and H. Geffner, 'Searching for Plans with Carefully Designed Probes', in *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*, (2011).

[20] James MacGlashan, 'Hierarchical Skill Learning for High-Level Planning', in *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10)*, (2010).

[21] B. Marthi, S. Russell, and J. Wolfe, 'Angelic Hierarchical Planning: Optimal and Online Algorithms', in *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, pp. 222–231, (2008).

[22] M. Martin and H. Geffner, 'Learning Generalized Policies in Planning Using Concept Languages', in *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pp. 667–677, (2000).

[23] A. Menif, C. Guettier, and T. Cazenave, 'Planning and Execution Control Architecture for Infantry Serious Gaming', in *Proceedings of the 3rd International Planning in Games Workshop (PG'13)*, pp. 31–34, (2013).

[24] C. Miller, R. Goldman, H. Funk, P. Wu, and B. Pate, 'A Playbook Approach to Variable Autonomy Control: Application for Control of Multiple, Heterogeneous Unmanned Air Vehicles', in *Annual Meeting of the American Helicopter Society*, (2004).

[25] C. Muise, S. McIlraith, J. Baier, and M. Reimer, 'Exploiting N-Gram Analysis to Predict Operator Sequences', in *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, (2009).

[26] H. Munoz-Avila, D. Aha, L. Breslow, and D. Nau, 'HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations', in *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pp. 870–875, (1999).

[27] D. Nau, T. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman, 'SHOP2: An HTN Planning System', *Journal of Artificial Intelligence Research*, **20**, 379–404, (2003).

[28] M. Hakim Newton, J. Levine, M. Fox, and D. Long, 'Learning Macro-Actions for Arbitrary Planners and Domains', in *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pp. 256–263, (2007).

[29] I. Sánchez-Garzón, J. Fernández-Olivares, and L. Castillo, 'An Approach for Representing and Managing Medical Exceptions in Care Pathways Based on Temporal Hierarchical Planning Techniques', in *Process Support and Knowledge Representation in Health Care (ProHealth'12), Lecture Notes in Computer Science 7738*, pp. 168–182, (2013).

[30] W. van der Sterren, 'Multi-Unit Planning with HTN and A*', in *AIGameDev Paris Game AI Conference*, (2009).

[31] Nathaniel Waisbrot, Ugur Kuter, and Tolga Könik, 'Combining heuristic search with hierarchical task-network planning: A preliminary report.', in *FLAIRS Conference*, pp. 577–578, (2008).

[32] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, 'Automating DAML-S Web Services Composition Using SHOP2', in *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, pp. 195–210, (2003).

[33] S. Yoon, A. Fern, and R. Givan, 'Inductive Policy Selection for First-Order Markov Decision Processes', in *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI'02)*, pp. 568–576, (2002).

[34] S. Yoon, A. Fern, and R. Givan, 'Learning Control Knowledge for Forward Search Planning', *Journal of Machine Learning Research*, **9**, 683–718, (2008).

[35] H. Zhuo, D. Hu, C. Hogg, Q. Yang, and H. Munoz-Avila, 'Learning HTN Method Preconditions and Action Models from Partial Observations', in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pp. 1804–1809, (2009).