

Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way*

Ron Alford¹

Ugur Kuter²

Dana Nau^{1,3,2}

¹Department of Computer Science, ²Institute for Advanced Computer Studies,

³Institute for Systems Research

University of Maryland, College Park, Maryland 20742, USA

{ronwalf, ukuter, nau}@cs.umd.edu

Abstract

We show how to translate HTN domain descriptions (if they satisfy certain restrictions) into PDDL so that they can be used by classical planners. We provide correctness results for our translation algorithm, and show that it runs in linear time and space. We also show that even small and incomplete amounts of HTN knowledge, when translated into PDDL using our algorithm, can greatly improve a classical planner's performance. In experiments on several thousand randomly generated problems in three different planning domains, such knowledge speeded up the well-known Fast-Forward planner by several orders of magnitude, and enabled it to solve much larger problems than it could otherwise solve.

1 Introduction

Most classical planners are either *domain-independent*, hence work in any classical planning domain, or *domain-configurable*, hence can exploit domain-specific knowledge. Each approach has advantages and disadvantages:

- *Domain-configurable* planners can read and make use of domain-specific planning knowledge, typically in the form of control rules (e.g., TLPlan [Bacchus and Kabanza, 2000] and TALplanner [Kvarnström and Doherty, 2001]) or HTN methods (e.g., SIPE-2 [Wilkins, 1988], O-PLAN [Currie and Tate, 1991], SHOP [Nau *et al.*, 1999], and SHOP2 [Nau *et al.*, 2003]). Using this domain knowledge, such planners can solve much larger planning problems, and solve them more quickly, than domain-independent planners. But the domain-specific knowledge used by these planners can sometimes be quite complicated, hence difficult for the general user to specify.
- *Domain-independent planners*, such as FastForward (FF) [Hoffmann and Nebel, 2001], AltAlt [Nguyen *et al.*, 2002], SGPlan [Chen *et al.*, 2006], HSP [Bonet and

Geffner, 1999], Fast Downward [Helmert, 2006], and LPG [Gerevini *et al.*, 2003], usually do not need any domain-specific knowledge other than the planning operators for a domain. This makes it much easier for the general user to specify the input to these planners. On the other hand, a domain-independent planner may perform much worse in a given domain than a domain-configurable planner that has been given a good set of domain knowledge.

We show that HTN planning knowledge, if it satisfies some restrictions, can automatically be translated into PDDL, and that *even small amounts of such knowledge can greatly improve a classical planner's performance*. In particular:

- We describe how to translate a restricted class of HTN methods and operators into PDDL. We provide theorems showing that our translation is correct, that its time and space complexity are both linear, and that it can be used even on *partial* HTN models of a domain (which can be much easier to write than full HTN models).
- Our experiments show that by translating partial HTN models into PDDL, we can substantially improve a classical planner's performance. In experiments with the well-known Fast-Forward (FF) planner [Hoffmann and Nebel, 2001] on more than 3500 planning problems, the translated knowledge improved FF's running time by several orders of magnitude, and enabled it to solve much larger planning problems than it could otherwise solve.

2 Basic Definitions and Notation

Classical Planning. Our definitions for classical planning are based on the ones in [Ghallab *et al.*, 2004].

Let L be the set of all literals in a function-free first-order language. A *state* is any set of ground atoms of L . A *classical planning problem* is a triple $P = (s_0, g, O)$, where s_0 is the *initial state*, g is the *goal* (a set of ground literals of L), and O is a set of operators. Each operator $o \in O$ is a triple

$$o = (\text{name}(o), \text{precond}(o), \text{effects}(o)),$$

where $\text{name}(o)$ is o 's name and argument list, and $\text{precond}(o)$ and $\text{effects}(o)$ are sets of literals called o 's *preconditions* and *effects*. An *action* α is a ground instance of an operator. If a state s satisfies $\text{precond}(\alpha)$, then α is *executable* in s , producing the state $\gamma(s, \alpha) = (s - \{\text{all negated atoms in}$

*This work supported in part by AFOSR grant FA95500610405, NAVAIR contract N6133906C0149, DARPA's Transfer Learning Program, DARPA IPTO grant FA8650-06-C-7606, and NSF grant IIS0412812. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

$\text{effects}(\alpha)\} \cup \{\text{all non-negated atoms in } \text{effects}(\alpha)\}$. A *plan* is a sequence $\pi = \langle \alpha_1, \dots, \alpha_n \rangle$ of actions. π is a *solution* for P if, starting in s_0 , the actions are executable in the order given and the final outcome is a state s_n that satisfies g .

TSTN Planning. [Ghallab *et al.*, 2004] describes a restricted case of HTN planning called *Total-order Simple Task Network* planning, which we'll abbreviate as TSTN Planning. The definitions are as follows.

A *task* is a symbolic representation of an activity. Syntactically, it is an expression $\tau = t(x_1, \dots, x_q)$ where t is a symbol called τ 's *name*, and each x_i is either a variable or a constant symbol. If t is also the name of an operator, then τ is *primitive*; otherwise τ is *nonprimitive*. Intuitively, primitive tasks can be instantiated into actions, and nonprimitive tasks need to be decomposed (see below) into subtasks.

A *method* is a prescription for how to decompose a task into subtasks. Syntactically, it is a four-tuple

$$m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)),$$

where $\text{name}(m)$ is m 's name and argument list, $\text{task}(m)$ is the task m can decompose, $\text{precond}(m)$ is a set of preconditions, and $\text{subtasks}(m) = \langle t_1, \dots, t_j \rangle$ is the sequence of subtasks.

A *TSTN planning problem* is a four-tuple $P = (s_0, T_0, O, M)$, where s_0 is an initial state, O is a set of operators, T_0 is a sequence of ground tasks called the *initial task list*, and M is a set of methods.

If T_0 is empty, then P 's only solution is the empty plan $\pi = \langle \rangle$, and π 's *derivation* (the sequence of actions and method instances used to produce π) is $\delta = \langle \rangle$. If T_0 is nonempty (i.e., $T_0 = \langle t_1, \dots, t_k \rangle$ for some $k > 0$), then let $T' = \langle t_2, \dots, t_k \rangle$. If t_1 is primitive and there is an executable action α with $\text{name}(\alpha) = t_1$, then let $s_1 = \gamma(s_0, \alpha)$. If $P' = (s_1, T', O, M)$ has a solution π with derivation δ , then the plan $\alpha \bullet \pi$ is a solution to P (where $\bullet$ is concatenation) whose derivation is $\alpha \bullet \delta$. If t_1 is nonprimitive and there is a method instance m such that $\text{task}(m) = t_1$, and if s_0 satisfies $\text{precond}(m)$, and if $P' = (s_1, \text{subtasks}(m) \bullet T', O, M)$ has a solution π with derivation δ , then π is a solution to P and its derivation is $m \bullet \delta$.

Using TSTN Planners for Classical Planning. To use a TSTN planner in a classical planning domain D (i.e., a set of classical planning problems that all have the same operator set O), the usual approach is augment D with a set M of methods and a way to translate each classical goal g into a task list T_0^g . This maps each classical planning problem $P = (s_0, g, O)$ in D into a TSTN planning problem $P' = (s_0, T_0^g, O, M)$. The mapping is *correct* if P' is solvable whenever P is, and if the solutions for P' are also solutions for P . Since the objective is for P' to have a small search space, the set of solutions for P' may be much smaller than the set of solutions for P .

In the above mapping, we will say that M is *O-complete* if every operator in $o \in O$ is *mentioned* in M , i.e., at least one method in M has a subtask that is an instance of $\text{name}(o)$.

3 Translating TSTN to Classical

Let $P = (s_0, T, O, M)$ be a TSTN planning problem, and suppose T_0 is a correct translation (as defined above) of a

classical goal g . We now describe how to translate P (if a restriction holds) into a classical planning problem $\text{trans}(P) = (s'_0, g, O')$ that is *equivalent* to P in the following sense: as we'll show in Section 4, there is a one-to-one mapping from P 's solution derivations to $\text{trans}(P)$'s solutions.

Preliminaries. We begin by introducing a restriction. For every solution π of P , let the *non-tail height* of π be the number of levels of method decomposition used to produce π , ignoring tail decomposition (i.e., decomposition of the last task in a task list). Then either we need to extend the planning language to include function symbols,¹ or else we must be given an upper bound H on the non-tail height of all solutions of P .

We need the above restriction in order to implement a symbolic representation of a numeric counter, to keep track of the current number of levels of task decomposition. Here is how to implement the counter when H is given:²

- We'll introduce new constant symbols $d_0, d_1, \dots, d_H$ to denote levels of task decomposition, and a predicate symbol *level* so that the atom $\text{level}(d_i)$ can be used to mean that the current level of task decomposition is d_i . We give a special meaning to the constant symbol d_0 : it marks the successful end of method decomposition process.
- To specify a total ordering of the constant symbols, we will put new atoms $\text{next}(d_1, d_2)$, $\text{next}(d_2, d_3)$, $\dots$, $\text{next}(d_{H-1}, d_H)$ into the initial state.

In addition, for each method $m(x_1, \dots, x_k)$ and task $t(y_1, \dots, y_j)$ we will introduce new atoms $\text{do}_m(x_1, \dots, x_k)$ and $\text{do}_t(y_1, \dots, y_j)$.

Translating operators. Let o be any operator in O , and suppose $\text{name}(o) = o(x_1, \dots, x_n)$, $\text{precond}(o) = \{p_1, \dots, p_j\}$ and $\text{effects}(o) = \{e_1, \dots, e_k\}$. If o is not mentioned in M (whence M is not O -complete), then $\text{trans}(o) = o$. Otherwise $\text{trans}(o)$ is the following operator o' , which is like o except that it is applicable only when do_o is true, and it decrements the counter:

$$\begin{aligned} \text{name}(o') &= o'(x_1, \dots, x_n) \\ \text{precond}(o') &= \{\text{do}_o(x_1, \dots, x_n), p_1, \dots, p_j, \\ &\quad \text{level}(v), \text{next}(u, v)\} \\ \text{effects}(o') &= \{-\text{do}_o(x_1, \dots, x_n), -\text{level}(v), \\ &\quad \text{level}(u), e_1, \dots, e_k\} \end{aligned}$$

We define $\text{trans}(O) = \{\text{trans}(o) \mid o \in O\}$.

Translating methods. Let m be any method in M , and suppose $\text{name}(m) = m(x_1, \dots, x_n)$, $\text{task}(m) = t(y_1, \dots, y_j)$, and $\text{precond}(m) = \{p_1, \dots, p_{j_m}\}$. There are two cases:

¹PDDL includes this extension, but traditional formulations of classical planning (e.g., [Ghallab *et al.*, 2004]) do not.

²If H is not given but the planning language contains function symbols, we can instead use an unbounded number of ground terms $1, \text{next}(1), \text{next}(\text{next}(1)), \dots$.

Case 1: $\text{subtasks}(m) = \emptyset$ (i.e., m specifies no subtasks for t). Then $\text{trans}(m)$ is the operator m' defined as follows:

$$\begin{aligned} \text{name}(m') &= m'(x_1, \dots, x_n) \\ \text{precond}(m') &= \{\text{do}_t(y_1, \dots, y_{j_t}), p_1, \dots, p_j, \\ &\quad \text{level}(v), \text{next}(u, v)\} \\ \text{effects}(m') &= \{\neg \text{do}_t(y_1, \dots, y_{j_t}), \neg \text{level}(v), \text{level}(u)\} \end{aligned}$$

Case 2: $\text{subtasks}(m) = \{t_1, \dots, t_k\}$ for $k \geq 1$. Then $\text{trans}(m)$ is the set of planning operators $\{m'_0, \dots, m'_k\}$ defined below, where m'_0 is an operator that checks whether m is applicable, and $m'_1, \dots, m'_k$ are operators that correspond to calling m 's subtasks. The definition of m'_0 is

$$\begin{aligned} \text{name}(m'_0) &= m'_0(x_1, \dots, x_n) \\ \text{precond}(m'_0) &= \{\text{do}_t(y_1, \dots, y_{j_t}), p_1, \dots, p_{j_m}, \text{level}(v)\} \\ \text{effects}(m'_0) &= \{\neg \text{do}_t(y_1, \dots, y_{j_t}), \text{do}_{m_1}(x_1, \dots, x_n, v)\} \end{aligned}$$

Intuitively, m'_0 's preconditions say that t is the current task and that m 's preconditions hold; and m'_0 's effect do_{m_1} makes it possible to apply the planning operator m'_1 that corresponds to m 's first subtask.

For $i = 1, \dots, k-1$, if m 's i^{th} subtask is $t_i(y_{i1}, \dots, y_{ij_i})$ then m'_i is defined as follows:

$$\begin{aligned} \text{name}(m'_i) &= m'_i(x_1, \dots, x_n) \\ \text{precond}(m'_i) &= \{\text{do}_{m_i}(x_1, \dots, x_n, v), \text{level}(v), \text{next}(v, w)\} \\ \text{effects}(m'_i) &= \{\neg \text{do}_{m_i}(x_1, \dots, x_n, v), \neg \text{level}(v), \text{level}(w), \\ &\quad \text{do}_{t_i}(y_{i1}, \dots, y_{ij_i}), \text{do}_{m_{i+1}}(x_1, \dots, x_n, v)\} \end{aligned}$$

The operator m'_k , which corresponds to m 's last subtask t_k , is like m'_i but omits the effects $\neg \text{level}(v)$ and $\text{level}(w)$.

We define $\text{trans}(M) = \bigcup_{m \in M} \text{trans}(m)$.

Translating planning problems. Finally, we define $\text{trans}(P) = (s'_0, g, O')$, where

$$\begin{aligned} s'_0 &= s_0 \cup \{\text{next}(d_0, d_1), \dots, \text{next}(d_{k-1}, d_k), \\ &\quad \text{level}(d_1), \text{do}_{t_0}(c_1, \dots, c_n)\}; \\ O' &= \text{trans}(O) \cup \text{trans}(M). \end{aligned}$$

4 Properties

The theorems in this section establish the correctness and computational complexity of our translation scheme.

Theorem 1 Let $P = (s_0, \langle t_1, \dots, t_k \rangle, O, M)$ (where $k \geq 0$) be any TSTN planning problem. Let $\Delta = \{\text{all derivations of solutions for } P\}$, and $\Pi = \{\text{all solutions for } \text{trans}(P)\}$. If M is O -complete, then there is a one-to-one correspondence that maps Δ onto Π .

Sketch of proof. We need to define a mapping $F : \Delta \rightarrow \Pi$ and show that F is one-to-one and onto. Below we define F ; the proof that it is one-to-one and onto can be done straightforwardly by induction.

Let π be a solution for P with derivation δ . Recall that δ is the sequence of the actions and method instances used to produce π , in the order that they were applied. In particular, δ is a concatenation of subsequences $\delta_1, \dots, \delta_k$ corresponding

to $t_1, \dots, t_k$. We will let $F(\delta) = F(\delta_1) \bullet \dots \bullet F(\delta_k)$, where $\bullet$ denotes concatenation, and where each $F(\delta_i)$ is defined recursively as follows:

If δ_i is empty, then $F(\delta_i)$ also is empty. If δ_i is nonempty (i.e., $\delta_i = \langle \alpha_{i1}, \dots, \alpha_{ik} \rangle$), then let $\delta'_i = \langle \alpha_{i2}, \dots, \alpha_{ik} \rangle$. There are three cases:

1. If α_{i1} is an action, then $F(\delta_i) = \text{trans}(\alpha_{i1}) \bullet F(\delta'_i)$.
2. If α_{i1} is a substitution instance $m\theta$ of a method m with substitution θ , and $\text{subtasks}(m)$ is empty, then $F(\delta_i) = m'\theta \bullet F(\delta'_i)$, where m' is as in Case 1 of Section 3.
3. If α_{i1} is a substitution instance $m\theta$ of a method m and $\text{subtasks}(m)$ is nonempty (i.e., $\text{subtasks}(m) = \langle t'_1, \dots, t'_j \rangle$ for some $j > 0$), then δ'_i is the concatenation of subsequences $\delta'_{i1}, \dots, \delta'_{ij}$ produced by decomposing $t'_1, \dots, t'_j$, respectively. In this case,

$$F(\delta_i) = m'_0\theta \bullet m'_1\theta \bullet F(\delta'_{i1}) \bullet \dots \bullet m'_j\theta \bullet F(\delta'_{ij}),$$

where $m'_1, \dots, m'_j$ are as in Case 2 of Section 3. $\square$

Corollary 1 In Theorem 1, if M is not O -complete, then the mapping F is one-to-one but not necessarily onto.

Sketch of proof. If M is not O -complete, then there is at least one operator $o \in O$ that is not mentioned in M . Consequently, no instance of o will appear in any solution for P , nor in Δ , hence no instance of $\text{trans}(o)$ will appear in $\{F(\delta) \mid \delta \in \Delta\}$. But instances of $\text{trans}(o)$ can appear in solutions to $\text{trans}(P)$, in which case F is no longer onto. $\square$

Theorem 2 The time and space complexity of computing $\text{trans}(P)$ are both $O(|P| + H)$.

Sketch of proof. For each $o \in O$, $\text{trans}(o)$ is a single operator that is computed by a linear-time scan of o , and it can be seen by inspection that the size of that operator is $O(|o|)$. Suppose there are no non-tail recursive methods in M . This means that $H = 0$ in this case. For each $m \in M$, $\text{trans}(m)$ is a set of methods that can be produced by a linear-time scan of m , and it can be seen by inspection that the set of methods has size $O(|m|)$. If there is a non-tail recursive method in M , then H is given as input and it is a fixed number. Thus, the theorem follows. $\square$

5 Implementation and Experiments

We implemented an algorithm that uses our translation technique to translate TSTN domain descriptions into PDDL, and did an experimental investigation of the following question:

In domains that are hard for a classical planner, how much can its performance be improved by PDDL translations of partial HTN knowledge?

For the classical planner, we used FF [Hoffmann and Nebel, 2001]. FF is perhaps one of the most influential classical planners available; many recent classical planning algorithms either directly depends on generalizations of FF or they incorporate the core ideas of FF in their systems.

For the planning domains, we chose three planning domains for which we wrote simple HTN domain descriptions with varying amounts of incompleteness: the Blocks World,

the Towers of Hanoi problem, and a transportation domain called the *Office Delivery* domain.

The source code for our translation technique and the HTN method descriptions of the three planning domains described below are available at <http://www.cs.umd.edu/projects/planning/data/alford09translating/>.

Towers of Hanoi. The Towers of Hanoi problem causes problems for many classical planners because of its combinatorial nature. On the other hand, it is almost trivially easy to write a set of HTN methods to solve the problem without any backtracking. The methods say basically the following:

- **Method to move a disk:**
precond: the smallest disk wasn't the last one moved
subtask: move the smallest disk clockwise.
- **Method to move a disk:**
precond: the smallest disk was the last one moved
subtask: move the other disk.

Note that in a tower of a Towers of Hanoi problem, the largest disk is always at the bottom of the tower and no disk can be placed on a smaller disk – i.e., the disks in a tower are in the increasing order by their sizes with the smallest is always at the top. Thus, whether the smallest disk was the last one moved can be checked in the above methods by examining the towers to the left or to the right of a tower.

The methods above provide an almost-complete solution to the Towers of Hanoi problem, except that the second method doesn't say where to move the disk. To use the PDDL translation, FF must figure out for itself that there is only one place the disk can be moved.

Below, “FF-Plain” refers to FF using the ordinary classical-planning definition of the Towers of Hanoi domain, and “FF-HTN” refers to FF using the PDDL translations of the HTNs described above. We varied the number n of disks from 3 to 14. For each value of n , we ran FF-HTN and FF-Plain each 100 times, averaging the running times. The reason for the multiple runs is because FF makes some random choices during each run that make its running time vary from one run to another. Fig. 1 shows the results.

For FF-Plain at 14 rings (the * in Fig. 1), two runs took longer than 2 hours (our time limit per problem) to finish. We counted these runs as 2 hours each, and averaged them with the other 98 runs; hence the data point for 14 rings makes FF-Plain's performance look better than it actually was.

As shown in Fig. 1, FF-Plain's running times grew much faster than FF-HTN's did. With 14 disks, FF-HTN was about 2 orders of magnitude faster than FF-Plain.

Blocks World. The Blocks World has previously been shown to pose some difficulties for FF. Complete HTN domain descriptions can work very efficiently [Ghallab *et al.*, 2004], but are somewhat complicated. To see how well FF could do with some simple and partial HTN knowledge, we wrote HTN methods that said basically the following:

- **Method to move a block:**
precond: the block is not in its final position
subtasks: pick up the block; put it in its final position.

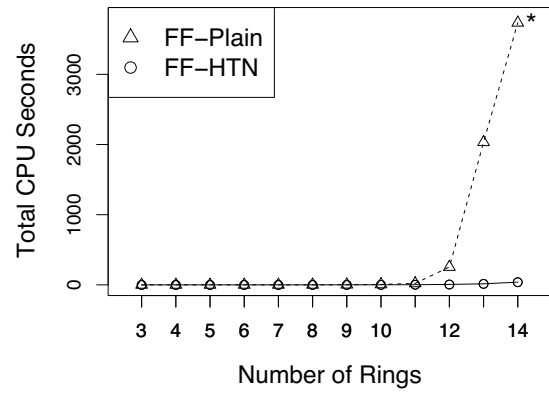


Figure 1: FF's CPU time in the Towers of Hanoi domain, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 runs. The asterisk is explained in the text.

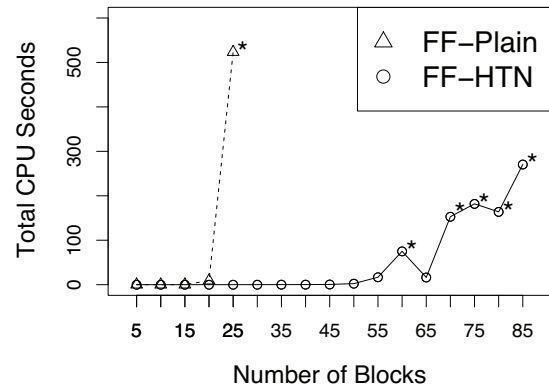


Figure 2: FF's CPU time in the Blocks World, with and without the translated domain knowledge. Each data point is FF's average CPU time on 100 randomly generated planning problems. The asterisks are explained in the text.

- **Method to move a block:**
precond: the block is not in its final position
subtasks: pick up the block; put it on the table.

At each point in the planning process, both of the methods are applicable. To use the PDDL translation of them, FF must use its heuristics to choose which of them to use.

Below, “FF-Plain” refers to FF using the ordinary classical-planning definition of the Blocks World, and “FF-HTN” refers to FF with the PDDL translations of the methods described above. We ran both FF-HTN and FF-Plain on 100 randomly generated n -block problems for each of $n = 5, 10, 15, \dots, 90$, giving a total of 1800 Blocks World problems. Fig. 2 shows the results.

As before, we gave FF a 2-hour time limit for each run. At data points where all 100 runs took less than 2 hours each, the data point was the average time per run. At data points where 3 or fewer of the 100 runs failed to finish within 2 hours, we counted each failure as 2 hours when computing the average, and marked the data point with an asterisk. In all of the other

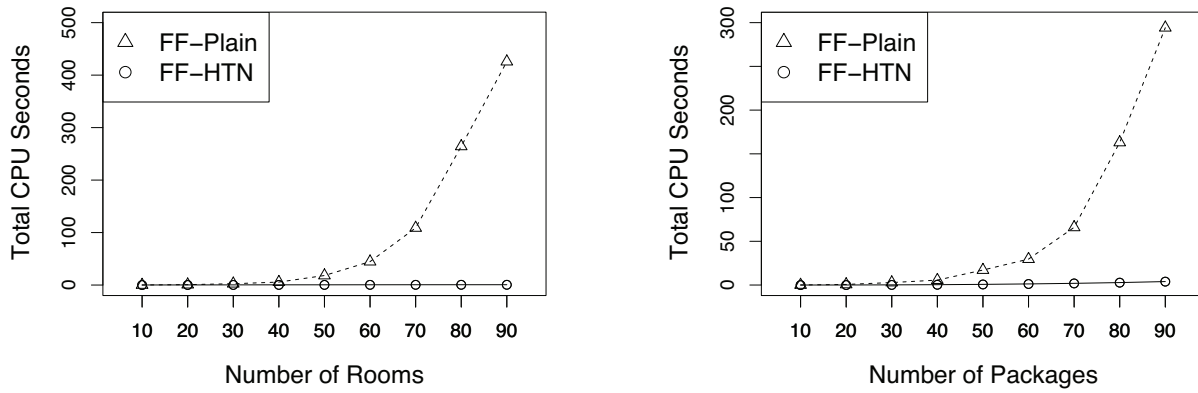


Figure 3: FF’s CPU time in the Office domain, with and without the translated domain knowledge. In the graph at left, the number of packages is fixed at 40 and the number of rooms varies. In the graph at right, the number of rooms is fixed at 40 and the number of packages varies. Each data point is FF’s average CPU time on 100 randomly generated planning problems.

cases, a large number of the 100 runs failed to finish with 2 hours, so we omitted those data points.

As shown in Fig. 2, FF-Plain could not solve problems larger than 25 blocks, but FF-HTN could solve problems up to 85 blocks. At 25 blocks, FF-HTN was about 4.2 orders of magnitude faster than FF-Plain.

Office Delivery. This is a transportation domain in which a robot needs to pick up and deliver packages in a building. It is similar to the well-known Robot Navigation Domain [Kabanza *et al.*, 1997], with the following differences: (1) the problem is deterministic, (2) there is a variable number of rooms, and (3) some of the rooms can be quite far from the hallway (hence to get to a room r , the robot may need to go through many other rooms). For this domain, we wrote a very incomplete set of HTN methods:

- **Method to move all remaining packages:**
 precondition: there is a package that’s not at its destination
 subtasks: pick up the package; put it in its final location;
 move all remaining packages.

Above, we omitted (1) how to get to the package’s location in order to pick it up, and (2) how to take the package to its destination. To use the PDDL translation of the method, the planner must figure out those things for itself.

Fig. 3 shows the results of our Office Delivery experiments. For the graph on the left, we fixed the number of packages at 40 and varied the number n of office rooms from 10 to 90; and for the graph on the right we fixed the number of rooms at 40 and varied the number k of packages from 10 to 90. For each combination of n and k we ran FF on 100 randomly generated problems, giving a total of 1700 problems.

As shown in Fig. 3, FF-Plain’s running time increased much faster than FF-HTN’s. On the largest problems (90 rooms and 90 packages), FF-HTN was faster than FF-Plain by about 2.8 and 1.9 orders of magnitude, respectively.

6 Related Work

Section 1 began with a discussion of related work on domain-independent and domain-configurable planners. We now dis-

cuss additional related work.

[Smith *et al.*, 2008] described the Action Notation Modeling Language (ANML) as an alternative to the existing planning languages such as PDDL and HTNs, while preserving a clear semantics. [Smith *et al.*, 2008] also describes a technique for translating HTNs specified as in ANML into PDDL. The ANML translation has several similarities to ours, but also a difference that can significantly affect planning time and solutions found: the ANML translation does not distinguish (see our “translating operators” subsection) between actions that aren’t used as subtasks of methods (in which case we use standard PDDL semantics) and actions that are (in which case we use standard HTN semantics, i.e., the action is applicable only when called by the methods that mention it). This affects both planning time and solutions found, and would seem to relate to the two sides of the controversy described at the start of “HTN Decomposition” section in [Smith *et al.*, 2008]. We believe there is merit in both sides of this controversy.

[Estlin *et al.*, 1997] argued that the knowledge-engineering effort required to produce effective HTN planning knowledge could be reduced by using partial-order-planning techniques such as causal-link analysis and goal regression, and using HTNs only to specify high-level goal hierarchies. They pointed out how more-specific but similar combinations of HTNs and classical-planning techniques were useful and effective in two planning systems developed at NASA.

In a similar vein, [Kambhampati *et al.*, 1998] proposed a plan-space refinement framework to allow HTN planning knowledge to be combined with classical planning, and argued that this would provide a principled way of handling partially hierarchical domains. As an instance of this approach, they cited [Mali and Kambhampati, 1998], which describes how to translate a restricted case of HTN planning into the satisfiability problem.

[Dix *et al.*, 2003] describes a translation of TSTN planning problems into *Answer Set Programs (ASPs)*. In their experiments, the approach did not perform as well as the SHOP planning algorithm [Nau *et al.*, 1999], but it provided substantial speedups compared to direct formulations of classical planning as ASPs.

[Baier *et al.*, 2007] provide a way to translate a subset of GOLOG into classical planning problems via finite state automata. The translation supports conditionals, loops and non-deterministic choice, but lacks procedures. [Fritz *et al.*, 2008] provides a theoretical extension which supports concurrency and procedures, and would support TSTN translation by first translating to ConGolog. In contrast, we provide a direct translation of TSTNs to PDDL, an implementation, and experimental results.

As a way to do planning with incomplete HTN knowledge, the Duet planner [Gerevini *et al.*, 2008] combines a domain-independent planner, LPG [Gerevini *et al.*, 2003], with a domain-configurable planner, SHOP2 [Nau *et al.*, 2003]. During planning, Duet uses SHOP2 to decompose tasks into smaller subtasks, and LPG to satisfy goal conditions. In their experiments, Duet was able to solve classical planning problems faster, on average, than LPG.

7 Conclusions

Our results show that HTN planning knowledge, if it satisfies the restrictions described in Section 3, can easily be translated into a form usable by domain-independent PDDL planners.

In our experiments with FF, PDDL translations of small amounts of HTN planning knowledge improved FF’s performance by several orders of magnitude. This occurred even though the HTN knowledge was *incomplete*, i.e., it omitted some of the knowledge that an HTN planner would need. In places where the knowledge was missing, FF simply used its ordinary planning heuristics.

FF’s ability to augment the translated HTN knowledge with its own heuristics suggests that if we were to take a complete domain description (e.g., for an HTN planner such as SHOP) and translate it into PDDL, this might enable FF (and perhaps other PDDL planners) to perform better than HTN planners. We haven’t yet been able to test this hypothesis since our current set of HTNs are only partial domain descriptions, but we hope to test it in the near future.

References

- [Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.
- [Baier *et al.*, 2007] J. Baier, C. Fritz, and S. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, 2007.
- [Bonet and Geffner, 1999] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *ECP*, 1999.
- [Chen *et al.*, 2006] Y. Chen, C. Hsu, and B. Wah. Temporal planning using subgoal partitioning and resolution in SGPlan. *JAIR*, 26:323–369, 2006.
- [Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.
- [Dix *et al.*, 2003] J. Dix, U. Kuter, and D. Nau. Planning in answer set programming using ordered task decomposition. In *KI*, 2003.
- [Estlin *et al.*, 1997] T. A. Estlin, Steve Chien, and X. Wang. An argument for a hybrid HTN/operator-based approach to planning. In *ECP*, 1997.
- [Fritz *et al.*, 2008] C. Fritz, J. Baier, and S. McIlraith. Congolog, sin trans: Compiling congolog into basic action theories for planning and beyond. In *KR*, 2008.
- [Gerevini *et al.*, 2003] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in lpg action graphs in LPG. *JAIR*, 20:239–290, 2003.
- [Gerevini *et al.*, 2008] A. Gerevini, U. Kuter, D. Nau, A. Saetti, and N. Waisbrot. Combining domain-independent planning and HTN planning: The Duet planner. In *ECAI*, 2008.
- [Ghallab *et al.*, 2004] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
- [Helmert, 2006] M. Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.
- [Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
- [Kabanza *et al.*, 1997] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.
- [Kambhampati *et al.*, 1998] S. Kambhampati, A. Mali, and B. Srivastava. Hybrid planning for partially hierarchical domains. In *AAAI*, 1998.
- [Kvarnström and Doherty, 2001] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *AMAI*, 30:119–169, 2001.
- [Mali and Kambhampati, 1998] A. Mali and S. Kambhampati. Encoding HTN planning in propositional logic. In *AIPS*, 1998.
- [Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *IJCAI*, 1999.
- [Nau *et al.*, 2003] D. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *JAIR*, 20:379–404, 2003.
- [Nguyen *et al.*, 2002] N. Nguyen, S. Kambhampati, and R. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, 2002.
- [Smith *et al.*, 2008] David E. Smith, Jeremy Frank, and William Cushing. The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2008.
- [Wilkins, 1988] D. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.