Learning Hierarchical Task Networks from Plan Traces

Chad Hogg and **Hector Muñoz-Avila**
Lehigh University

Abstract

We present HTN-MAKER, an offline and incremental algorithm for learning the structural relations between tasks in a Hierarchical Task Network (HTN). HTN-MAKER receives as input a STRIPS domain model, a collection of STRIPS plans, and a collection of task definitions, and produces an HTN domain model. HTN-MAKER is capable of learning an HTN domain model that reflects the provided task definitions. In particular, if the tasks have different levels of abstraction, these will be reflected in the HTN. We have conducted an empirical evaluation of HTN-MAKER on the logistics-transportation domain. These experiments demonstrate that HTN-MAKER quickly learns an HTN domain model that may be used to solve nearly all problems in the domain. Challenges and future work are discussed.

Introduction

Hierarchical Task Network (HTN) planning is an important, frequently studied research topic in artificial intelligence. Researchers have reported work on its formalisms and applications (Erol, Hendler, & Nau 1994; Smith, Nau, & Erol 1998; Nau *et al.* 2005). In HTN planning, complex tasks are decomposed into simpler tasks until a sequence of primitive actions is generated. There are three main motivations for the recurrent interest in HTN planning. First, researchers have pointed out that one way to model how humans acquire knowledge is through a hierarchy of skills. Humans begin by learning simpler tasks and then proceed by learning more complex tasks that build on existing knowledge. Thus, hierarchical modeling is at the core of many cognitive architectures (Choi & Langley 2005). Second, HTN planning is a natural representation for many real-world domains such as computer games (Smith, Nau, & Erol 1998) and storytelling (Cavazza & Charles 2005). Third, HTN planning has played a fundamental role in the remarkable advances of AI planning research over the past few years. HTN knowledge representation principles of capturing domain-specific strategies for problem-solving while performing domain-independent search was in part the motivation for the so-called domain-configurable planners such as SHOP (Nau *et al.* 1999), which have demonstrated impressive speed gains over earlier classical (STRIPS) planners. These new paradigms for planning have improved the runtime performance for solving planning problems by several orders of magnitude.

Despite these successes, a major hurdle for the use of HTN planning is the need for an HTN domain description. In fact, a controversy in the AI planning research community surrounds the recent efficiency gains obtained with HTN planning because the domain descriptions sketch the underpinnings of the solutions. Therefore, it has been argued that a significant knowledge engineering effort is required to obtain such domain descriptions. A domain description is a collection of knowledge constructs describing the target domain. In HTN planning, a domain description consists of the action model and the task model. The action model encodes knowledge about valid actions or primitive tasks changing the world state. The task model encodes knowledge about how to decompose tasks into subtasks, and is the part of the domain description that has been argued to be difficult to obtain. Given the large interest in HTN planning, it is surprising that little research has been done on learning task models. The bulk of research involving planning and learning has focused on search control knowledge (Zimmerman & Kambhampati 2003).

We present HTN-MAKER (Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required), an offline and incremental algorithm for learning task models. HTN-MAKER receives as input a collection of plans generated by a STRIPS planner, an action model, and a collection of task definitions, and it produces a task model. When combined with the action model, this task model results in an HTN domain model that may be used by an HTN planner to solve problems in the domain. HTN planning with this domain model is sound but not necessarily complete. That is to say that there may be problems that could be solved by a STRIPS planner using the action model alone but that cannot be solved by an HTN planner using the learned HTN domain model. However, any plans generated from the HTN domain model will be correct in terms of the original STRIPS model.

We have performed an evaluation of HTN-MAKER on the logistics-transportation domain and found that HTN-MAKER is successful in learning an HTN domain model capable of solving nearly all problems in the domain based on a few examples. However, the over-generality of the learned task models is a challenging open problem in most other domains.

Related Research

Learning task decompositions means eliciting the hierarchical structure relating tasks and subtasks. Existing work on learning hierarchies elicits a hierarchy from a collection of plans and from a given action model (Choi & Langley 2005; Reddy & Tadepalli 1997; Ruby & Kibler 1991). A particularity of the existing work on learning task models is that the tasks from the learned hierarchies are the same goals that have been achieved by the plans. Reddy and Tedepally's 1997 X-Learn, for example, uses inductive generalization to learn task decomposition constructs, which relate goals, subgoals, and conditions for applying d-rules. By grouping goals in this way, task models are learned that lead to speed-up in problem-solving. However, it is possible to solve the same problems without the learned task models. In the experiments reported in (Choi & Langley 2005; Reddy & Tadepalli 1997) some of the problems that were solved using the learned task models could not be solved without using them (e.g., by using only the action models). This is due to the speed-up gains that allow these systems to obtain solutions in the pre-defined time. However, these problems could theoretically be solved using the action models alone if the search was performed systematically and enough time was given.

Two recent studies (Ilghami *et al.* 2005; Xu & Munoz-Avila 2005) propose eager and lazy learning methods respectively to learn the preconditions of HTN methods. These systems require as input the hierarchical relationships between tasks and learn only the conditions under which a method may be used. Another recent work by Langley & Choi 2005 learns a special case of HTNs known as teleoreactive logic programs. Rather than a task list, this system uses a collection of Horn clause-like concepts. The means-end reasoning that is tightly integrated with this learning mechanism is known to be incapable of solving some problems that general HTNs are able to solve, such as the register assignment problem.

Although the state of the art in learning task models has resulted in speed-up gains for problem-solving, the learned task models are far from the kinds of task models that motivated HTN planning. In almost any description of HTN planning found in the literature it is said that highly complex tasks should be decomposed into less complex tasks, which are further decomposed eventually into low-level tasks (e.g., (Erol, Hendler, & Nau 1994; Nau *et al.* 1999)). For example, in the blocks world domain, higher level tasks could represent actions on piles of blocks, whereas lower level tasks represent manipulations of pairs of blocks. In existing algorithms for learning task models, all tasks represent pairs of blocks because these are goals used in the classic action model of the blocks world.

The principle of representing tasks of increasing complexity at higher levels of a hierarchy is by no means unique to HTN planning. In abstraction planning the same principle is followed; Bergmann & Wilke 1995 demonstrate how goals at higher levels of a hierarchy may be expressed in a different, more abstract language than goals at more concrete levels. A similar point is made for cognitive architectures; the concepts expressed at higher levels are at a different level of granularity than concepts at the concrete level.

Work on learning macro-operators (e.g., (Mooney 1988; Botea, Muller, & Schaeffer 2005)) falls in the category of speed-up learning, as do work on learning search control knowledge ((e.g., (Mitchell, Keller, & Kedar-Cabelli 1986; Minton 1998; Fern, Yoon, & Givan 2004)). Search control knowledge does not increase the number of problems that theoretically can be solved. However, from a practical stand point, these systems increase the number of problems that can be solved because of the reduction in runtime. Other researchers assumed that hierarchies are given as inputs for learning task models. (Garland, Ryall, & Rich 2001) uses interactive elicitation in which the user provides examples showing how to correctly perform a task and annotates other ways to perform the task in the examples.

Inductive approaches have been proposed for learning action models (e.g., (Martin & Geffner 2000; Winner & Veloso 2003)). For example, the DISTILL system learns domain-specific planners from an input of plans that have certain annotations (Winner & Veloso 2003). The input includes the initial state and an action model. DISTILL elicits a programming construct for plan generation that combines the action model and search control strategies.

Another related work is abstraction in planning such as the Alpine (Knoblock 1993) and the Paris (Bergmann & Wilke 1995) systems. These systems take a concrete plan and generalize it. This allows the reuse of the generalized plan in different problems by instantiating its conditions. These systems require both an action model and an abstraction model that indicates how to abstract and specialize plans.

Learning Hierarchical Relations From Tasks

We will first specify the problem of extracting hierarchies. In previous work for learning task hierarchies, tasks are goals and therefore the semantics of the learned hierarchies were clear. Given a hierarchy H with a goal g at the top level, the plan P obtained by collecting the actions in the leaves of H must achieve g to be correct. That is, one can examine the plan, regardless of the hierarchy, to determine if it is correct. This is not the case in general HTN planning. Informally, a plan is correct if an HTN exists that decomposes the top-level task(s) of the problem such that the HTN entails the plan. The top-level tasks represent complex goals that may not be in the vocabulary of the preconditions and effects of the actions in the plans. This means that the only way to verify if a plan is correct is by finding an HTN that entails it. This poses a problem for defining the kinds of tasks that are given in the task taxonomy so that the semantics of the resulting hierarchy unambiguously relates to the input problem-solution plan pairs.

Task Definitions

To address this problem, we have adopted the definition of tasks from process models. Loosely speaking, a process is the means by which tasks are accomplished via a series of actions or operations. Process models represent concept reuse and modification. They have been used in a wide range

```
( :task                        ( :task
  ( DELIVER ?obj ?dst )          ( INVERT-3PILE ?a ?b ?c )
  ( ( OBJ ?obj )                 ( ( on ?a ?b )
    ( LOCATION ?dst ) )            ( on ?b ?c ) )
  ( ( at ?obj ?dst ) )           ( ( on ?c ?b )
)                                  ( on ?b ?a ) )
                               )
```

Table 1: Example Tasks

of applications, including to ease user interactions for AI planning systems (Fdez-Olivares *et al.* 2006) and to build introspective agents that use models to identify reasoning failures (Murdock 2001). In particular, we chose the task-method-knowledge (TMK) variant of process models. In TMKs, **tasks** indicate what they accomplish by stating their preconditions and effects. Task semantics are the following: if the preconditions are true in the state of the world and the task is accomplished, the effects must be true in the resulting world state.

Table 1 shows examples of tasks in the logistics-transportation and blocks-world domains. Each task description contains the task name and parameters, a list of preconditions, and a list of effects. The first task is used to deliver a package in the logistsics-transportation domain. The only preconditions for this task require that the parameters be of appropriate types. The effect of this task is to cause the package to be at the required destination. The second task is used to invert a stack of 3 blocks in the blocks-world domain. The preconditions here ensure that the parameters do in fact refer to a pile of 3 blocks, and the effect is to cause them to be piled in opposite order. It should be noted that the task descriptions do not specify how they may be accomplished; they only indicate what they accomplish. An HTN method, the construct learned by HTN-MAKER, specifies how to accomplish a given task.

The task definitions used as input for the hierarchy learning problem consist of a collection of tasks in this form. These become the nonprimitive tasks of the domain. The heads of the operators in the action model form the primitive tasks in the domain. Creation of these task definitions is not a significant burden; they simply describe the things that a planner might be asked to accomplish. Complete preconditions and effects are not necessary. Only the subset of effects that define the goal of a task are required, although other effects and preconditions may be used to make the process more efficient.

Learning Problem

The **task model learning problem** is defined as follows: given a collection of task definitions, a collection of STRIPS problems, a collection of plans solving these problems, and the action model used to generate these plans, obtain a task model. Under these preconditions and given a learned task model, one can check if a plan P **correctly** solves an HTN planning problem where $t_1 \ldots t_n$ are the tasks to achieve, and S is the initial state. To do this, one checks if the preconditions of t_1 are satisfied in S and if the effects of t_1 are satisfied in a state S_1. The state S_1 is obtained by executing the plan P_1 on S, where P_1 is the plan entailed by the portion of the HTN that accomplishes t_1. One can continue by checking task t_2 starting from S_1, and so forth for the remaining tasks.

The HTN-MAKER Algorithm

The HTN-MAKER algorithm traverses forward through a STRIPS plan, generating the new state after each action by applying it to the previous state. For each substitution of variables such that the current state includes all effects of a task it is possible to learn a set of methods: one that encapsulates the previous operator, another for the previous two operators, and so on. The methods that encapsulate a shorter section of the plan are learned first, so that they may be used as subtasks in methods that encapsulate longer sections of the plan.

Operators and previously learned methods may be added as subtasks of the new method, from last to first. An operator or method is chosen as a subtask if its effects provide either an effect of the task or a precondition of a later subtask. If no operator or method is useful at a particular step of the plan, that operator will be skipped as irrelevant to the method.

The learned method has the associated task as its head; the union of the task preconditions, preconditions of subtasks that are not satisfied by an earlier subtask, and effects of the task that are not satisfied by a subtask as its preconditions; and the collected operators and methods as its subtasks. This process is a variant of goal regression (Mitchell, Keller, & Kedar-Cabelli 1986) in which conditions are regressed both horizontally from one operator to the next and vertically up the task hierarchy.

Algorithm 1 is a very relaxed pseudocode of HTN-MAKER. The pseudocode abstracts away difficulties of finding a proper substitution for each subtask and unifying them with each other and does not provide such necessary features as building multi-level hierarchies in which methods have non-primitive subtasks. Nevertheless, it should provide enough detail to understand the general operation of HTN-MAKER.

The outermost loop (line 4) iterates forward through the states created by the plan. Within each state s, there is a search for all tasks that have their effects satisfied in that state (lines 5 and 6). In the actual implementation, the task definitions contain variables and therefore considers all possible substitutions that make the inclusion (line 6) true. We then consider subsequences of the plans to determine whether or not a method may be created from them, starting with the subsequence that contains only the operator that caused the current state, then adding the previous and so forth. The initial state in this subsequence is controlled by *init* (line 7), while the last state in the subsequence is always s. We maintain a list of remaining effects to be achieved, which is initially all of the task effects (line 8), a list of remaining preconditions to be achieved, which is initially empty (line 9), and a list of subtasks, which is initially empty (line 10). The inner-most loop (lines 11-16) walk backward through the subsequence, adding operators to the subtasks list. Any effects of a selected operator are removed from

the list of remaining effects (line 14) and remaining preconditions (line 15), and then the preconditions of the operator are added to the remaining preconditions (line 16). The implementation handles substitutions unifiying variables from the various subtasks with those of the task definition, ignores operators that do not contribute to the remaining effects or preconditions, and may select a previously learned method rather than one or more operators. Finally, a new method is created (line 18) with the task as its head, the collected operators and methods as its subtasks, and the union of task preconditions, task effects that were not achieved by a subtask, and outstanding subtask preconditions. This method is added to the domain (line 19), although the implementation will decline to add some methods for reasons discussed in the section on Open Problems and Future Work.

Algorithm 1 HTN-MAKER(P, D, T)

1: **Input**: P is a plan, consisting of actions and initial, final, and intermediate states; D is a partial HTN domain description; T is a set of task definitions
2: **Output**: D' is an enhanced HTN domain description that can be used to generate P
3: $D' \leftarrow D$
4: **for** $s \leftarrow$ first state of P to last state of P **do**
5:   **for all** tasks t in T **do**
6:     **if** effects of $t \subseteq s$ **then**
7:       **for** $init \leftarrow s$ downto first state of P **do**
8:         $rem\text{-}effects \leftarrow$ effects of t
9:         $rem\text{-}precs \leftarrow \emptyset$
10:         $subtasks \leftarrow \emptyset$
11:         **for** $current \leftarrow s$ downto $init$ **do**
12:           $oper \leftarrow$ operator causing $current$
13:           prepend $oper$ to $subtasks$
14:           $rem\text{-}precs \leftarrow rem\text{-}precs \setminus$ effects of $oper$
15:           $rem\text{-}effects \leftarrow rem\text{-}effects \setminus$ effects of $oper$
16:           $rem\text{-}precs \leftarrow rem\text{-}precs \cup$ precs of $oper$
17:         $method\text{-}precs \leftarrow rem\text{-}precs \cup rem\text{-}effects \cup$ precs of t
18:         $new \leftarrow$ METHOD$(t, method\text{-}precs, subtasks)$
19:         $D' \leftarrow D' \cup \{new\}$
20: **return** D'

Example

Figure 1 exemplifies a resulting HTN for the logistics-transportation domain. The initial state in this case consists of a package *p1* in a truck *t1* at an airport *l1* that contains an airplane *a1*, and the goal is to deliver the package to a different airport *l2*. The plan consists of four actions: *unload-truck(p1,t1,l1)*, *load-plane(p1,a1,l1)*, *fly-plane(a1,l1,l2)*, and *unload-plane(p1,a1,l2)*.

Suppose that there is a single task, *DELIVER-PKG(?p,?l)*, with preconditions that *?p* be a package and *?l* be a location, and effects that *?p* be at *?l*. After the first operator, package *p1* has been delivered to location *l1*. Thus, HTN-MAKER will learn a method for solving this task bound to these constants. The operator *unload-truck(p1,t1,l1)* produces the effect *at(p1,l1)*, so it will be selected as a subtask. The learned
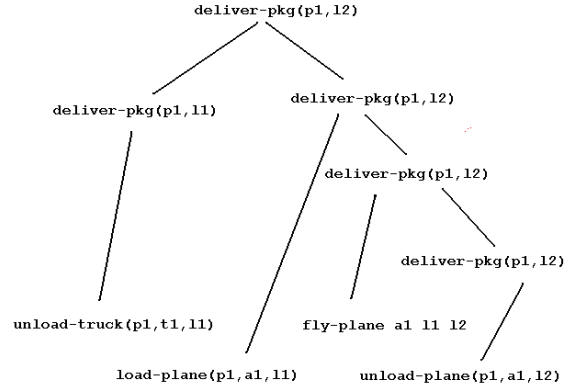

Figure 1: Example Plan Trace

method will be applicable when the types of variables are correct (from the task preconditions), and the package is in a truck that is at the destination (from the preconditions of the operator). In the next two states, there are no valid instantiations of the task effects.

In the final state, the package *p1* has been delivered to location *l2*. A recursive series of methods is learned. The first delivers a package that is in an airplane at the destination by unloading the airplane. The next delivers a package that is in an airplane at the wrong location by flying to the destination, which must be an airport, and then delivering. The third requires that the package be at an airport that is not the destination and that contains an airplane, and proceeds by loading the package into airplane and then continuing to deliver. The final first delivers to an airport, and then from there to the final destination.

An HTN planner presented with this initial state, goal, and collection of methods might build the same hierarchical structure from the top task down to the primitive actions. With a different initial state or goal, an HTN planner might use pieces of this structure integrated with other methods learned from other problems.

Empirical Validation

Our experimental hypothesis is that the output of HTN-MAKER on a given domain will converge to an HTN domain model that is able to solve nearly all solvable problems in the domain after a few problems are analyzed. We measure performance by the cummulative number of problems successfully solved as they are presented sequentially.

Experiment Setup

We slightly modified the logistics-transportation action model from the FF Domain Collection to prevent trucks from being driven from one location to itself and similarly for airplanes. This change merely improves the runtime of planners as they will no longer consider applying these actions with no effects. We generated 100 random problems with between 3 and 5 cities, each of which contains a truck,

an airport, and between 2 and 4 other locations, with a total of between 1 and 3 airplanes distributed among the airports and between 1 and 4 packages. The goals for each problem consist of transporting each package to a destination. The STRIPS planner Fast-Forward (Hoffmann & Nebel 2001) was used to generate a solution to each solvable problem.

We also developed a set of tasks for which methods could be learned. Although HTN-MAKER is capable of learning methods for accomplishing any type of task in the domain, we chose a task set for this experiment that reflects only the goals specified in the problems. This allows us to evaluate the learning algorithm by its ability to solve the same problems from which it learns.

We then completed 4 trials in which HTN-MAKER was used to learn an HTN domain model starting with an action model and an empty task model. Each trial began with such a base HTN domain model and processed the 100 problems in a random order. For each problem, we tried to solve it using the HTN domain model learned thusfar in the sound and complete HTN planner SHOP. If SHOP was able to solve the problem, we counted this problem as a success. Otherwise, the problem was counted as a failure and HTN-MAKER was used to update the current HTN domain model with methods learned by analyzing Fast-Forward's solution to the problem.

Results

The cummulative number of problems solved, averaged over the 4 trials is shown in Figure 2. After learning from a few initial failures, this line rapidly approaches the optimal $y = x$, where all problems are solved without the need for learning. On average, only 5.75 problems are not successfully solved by the HTN planner. The first 10 problems encountered contain 3.75 of these failures, while only 2.0 of the remaining 90 cannot be solved. Thus, our hypothesis is valid for the logistics-transportation domain. All of the methods used to solve the 94.25 solved problems were learned from STRIPS plan traces of the 5.75 failed problems. On average, 73 distinct methods are learned during the process. If the problems were presented in a designed order, we would expect even better results.

Open Questions And Future Work

While we have been able to produce good results in the logistics-transportation domain, these do not translate well to other common domains such as blocks-world. In all cases the soundness of the learned domain description in terms of producing only valid STRIPS plans is guaranteed[1], but it is often far from optimal. Specifically, there are two significant difficulties. The first is the very large number of methods that will be learned and that must be considered by an HTN planner using the domain description. The second, more serious problem is the possibility for the planner to use methods in an infinitely recursive manner.

---

[1]The proof of correctness is omitted here, but follows from the way preconditions are effects are regressed through the learned methods
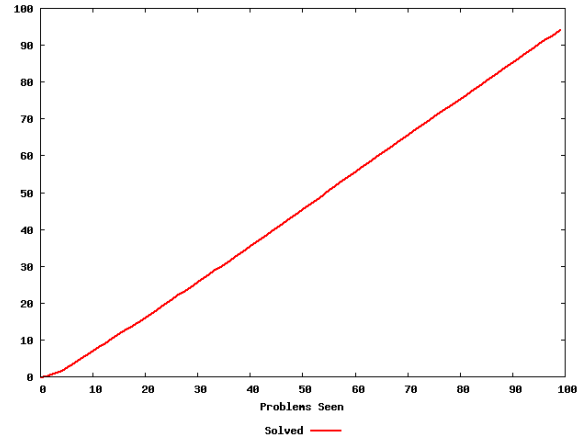

Figure 2: Cummulative Number of Logistics-Transportation Problems Solved

We have considered a number of steps to address the first difficulty. Primarily, whenever one learned method subsumes another, we retain only the most general method. We define a method a to subsume b if and only if there is a substitution μ such that subtasks of $a = \mu$(subtasks of b) and preconditions of $a \subseteq \mu$(preconditions of b). Thus, whenever b is applicable, a must also be. This reduces the number of methods somewhat and, because it replaces specific methods with general ones, should increase the number of opportunities for methods to be used in hierarchies dissimilar from those in which they were created.

> 1: *fly-plane(a1,l1,l2)*
> 2: *load-plane(p1,a1,l2)*
> 3: *fly-plane(a1,l2,l3)*
> 4: *load-plane(p2,a1,l3)*
> 5: *fly-plane(a1,l3,l4)*
> 6: *unload-plane(p1,a1,l4)*
> 7: *unload-plane(p2,a1,l4)*

Figure 3: Example Plan

As an example of the recursion problem, consider the logistics-transportation plan in Figure 3, where an airplane is used to deliver two packages from different locations to the same destination. Several methods will be learned for the task of delivering *p2*: one that encapsulates operator 7, one that concatenates operator 5 to the previous method, one that concatenates operator 4 to the method of 5 and 7, one that concatenates operator 3 to the method of 4, 5, and 7, and one that adds operator 1 to the method of 3, 4, 5, and 7. This last method is created because operator 1 provides *at(a1,l2)*, which is a precondition of the method that encapsulates operator 3. This method is dangerous because it allows the planner to fly the airplane to a location that is entirely irrelevant to the task, then do the same ad infinitum. This is a minor annoyance in the logistics-transportation domain, but recursive methods of a similar nature are learned in other

domains, and are much problematic. We are currently exploring several approaches to preventing these methods that allow allow infinite recursion from being learned.

Additionally, there is a nondeterministic choice to be made while learning a method between adding the operator as a subtask or instead adding a previously-learned method that encapsulates that operator. We initially envisioned taking each choice in turn, but found that this unnecessarily exploded the search space for the planner. For the experiments reported in this paper we chose a strategy that prefers to select the method that encapsulates the most operators when one exists, resulting in deep hierarchies. While this works well, it is not clear that it is the best approach.

The general question to be studied is the appropriate level of generality. A domain description that is too general allows infinite recursion and erodes the advantages of HTN planning over classical planning. A highly specific domain description is less likely to be capable of solving new problems and will require a much larger set of methods than should be necessary.

References

Bergmann, R., and Wilke, W. 1995. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research* 53–118.

Botea, A.; Muller, M.; and Schaeffer, J. 2005. Learning partial-order macros from solutions. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*. AAAI Press.

Cavazza, M., and Charles, F. 2005. Dialogue generation in character-based interactive storytelling. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press.

Choi, D., and Langley, P. 2005. Learning teleoreactive logic programs from problem solving. In *Proceedings of the Fifteenth International Conference on Inductive Logic Programming*. Springer.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Htn planning: complexity and expressivity. In *AAAI'94: Proceedings of the twelfth national conference on Artificial Intelligence (vol. 2)*, 1123–1128. Menlo Park, CA, USA: American Association for Artificial Intelligence.

Fdez-Olivares, J.; Castillo, L.; Garcia-Perez, O.; and Palao, F. 2006. Bringing users and planning technology together, experiences in siadex. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*. Menlo Park, CA: AAAI Press.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*. AAAI Press.

Garland, A.; Ryall, K.; and Rich, C. 2001. Learning hierarchical task models by defining and refining examples. In *Proceedings of the First International Conference on Knowledge Capture*, 363–391.

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

Ilghami, O.; Munoz-Avila, H.; Nau, D.; and Aha, D. W. 2005. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Knoblock, C. 1993. *Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning*. Norwell, MA: Kluwer Academic Publishers.

Martin, M., and Geffner, H. 2000. Learning generalized policies in planning using concept languages. In *Proceedings of the 7th Int. Conf. on Knowledge Representation and Reasoning (KR2000)*. Morgan Kaufmann.

Minton, S. 1998. *Learning Effective Search Control Knowledge: an Explanation-Based Approach*. Ph.D. Dissertation, Carnegie Mellon University.

Mitchell, T.; Keller, R.; and Kedar-Cabelli, S. 1986. Explanation-based generalization: A unifying view. *Machine Learning* 1.

Mooney, R. J. 1988. Generalizing the order of operators in macro-operators. *Machine Learning* 270–283.

Murdock, J. W. 2001. *Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation*. Ph.D. Dissertation, Georgia Institute Of Technology.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 968–973. AAAI Press.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Munoz-Avila, H.; Murdock, J. W.; Wu, D.; and Yaman, F. 2005. Applications of shop and shop2. *IEEE Intelligent Systems* 20(2):34–41.

Reddy, C., and Tadepalli, P. 1997. Learning goal-decomposition rules using exercises. In *Proceedings of the Internation Conference on Machine Learning (ICML-97)*.

Ruby, D., and Kibler, D. F. 1991. Steppingstone: An empirical and analytic evaluation. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 527–531. Morgan Kaufmann.

Smith, S. J. J.; Nau, D. S.; and Erol, K. 1998. Control strategies in htn planning: theory versus practice. In *IAAI '98: Proceedings of the tenth conference on Innovative applications of artificial intelligence*, 1127–1133. Menlo Park, CA, USA: American Association for Artificial Intelligence.

Winner, E., and Veloso, M. M. 2003. Distill: Learning domain-specific planners by example. In *Proceedings of the International Conference on Machine Learning*.

Xu, K., and Munoz-Avila, H. 2005. A domain-independent system for case-based task decomposition without domain theories. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*. AAAI Press.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* 73–96.