Computational Intelligence, Volume 21, Number 4, 2005

LEARNING PRECONDITIONS FOR PLANNING FROM PLAN TRACES AND HTN STRUCTURE

OKHTAY ILGHAMI
Department of Computer Science, University of Maryland

DANA S. NAU
Department of Computer Science and Institute for Systems Research, University of Maryland

HÉCTOR MUÑOZ-AVILA
Department of Computer Science and Engineering, Lehigh University

DAVID W. AHA
Navy Center for Applied Research in AI, Naval Research Laboratory (Code 5515), Washington, DC

A great challenge in developing planning systems for practical applications is the difficulty of acquiring the domain information needed to guide such systems. This paper describes a way to learn some of that knowledge. More specifically, the following points are discussed. (1) We introduce a theoretical basis for formally defining algorithms that learn preconditions for Hierarchical Task Network (HTN) methods. (2) We describe Candidate Elimination Method Learner (*CaMeL*), a supervised, eager, and incremental learning process for preconditions of HTN methods. We state and prove theorems about CaMeL's soundness, completeness, and convergence properties. (3) We present empirical results about CaMeL's convergence under various conditions. Among other things, CaMeL converges the fastest on the preconditions of the HTN methods that are needed the most often. Thus CaMeL's output can be useful even before it has fully converged.

Key words: HTN planning, learning, candidate elimination, version spaces.

1. INTRODUCTION

Artificial intelligence (AI) planning systems are finally becoming capable enough to be used in practical applications ranging from autonomous space vehicles (Muscettola et al. 1998) to robot control (Ghallab and Laruelle 1994) to computer bridge (Smith, Nau, and Throop 1998). One of the biggest obstacles to the development of such systems has been the difficulty of obtaining domain-specific problem-solving knowledge to help guide the planning system. Such information is essential to provide a satisfactory level of performance,[1] but it can be difficult to get human domain experts to spare enough time to provide information that is detailed and accurate enough to be useful, and it can be even harder to encode this information in the language the planner uses to represent its input.

Consequently, it is important to develop ways to learn the necessary domain-specific information automatically. In this paper, we propose and investigate a way for planning systems to learn some of it.

We focus specifically on Hierarchical Task Network (HTN) planning, which is the AI planning methodology that has gained the widest use in practical applications (Wilkins 1990; Currie and Tate 1991; Nau et al. 2004). The main knowledge artifacts for HTN planning are called *methods*. They indicate how to decompose high-level tasks into simpler tasks. Methods also indicate the *preconditions*, that is, the conditions under which such a decomposition is possible.

[1] An analogous difference in performance can be seen by comparing the performance of the "fully automated" and "hand-tailorable" planning systems in the last three International Planning Competitions (Bacchus 2000; Long and Fox 2002; Edelkamp and Hoffmann 2004).

Our contributions are as follows:

(1) We introduce a theoretical basis for formally defining algorithms that learn preconditions for HTN methods, and what soundness, completeness and convergence mean in this context. This formalism models situations in which

   (i) general information is available concerning the possible decompositions of tasks into subtasks, but without sufficient details to determine the conditions under which each decomposition will succeed;
   (ii) plan traces, each of which is known to be successful or unsuccessful, are available for certain problem instances.

(2) We present a supervised, eager, and incremental learning process for learning preconditions. The process is supervised by a domain expert who solves instances of the problems in that domain. Our algorithm, CaMeL (Candidate Elimination Method Learner), is based on a modified version of Candidate Elimination (Mitchell 1997). CaMeL keeps a version space for each HTN method in the domain it is learning, and it converges to a single HTN domain description given enough training data when each of these version spaces converges to a single node. We state and prove theorems about CaMeL's soundness, completeness, and convergence properties.

(3) We provide empirical results revealing the speed with which CaMeL converges in different situations. Among other things, our experiments show that CaMeL fully learns the preconditions of the HTN methods that are needed the most often much faster than the preconditions of less frequently used HTN methods. Thus, CaMeL's output is useful even before it has fully converged. This suggests that CaMeL may potentially be of use in real-world planning domains even if only a small number of training samples are available.

2. BACKGROUND

2.1. HTN Planning

In HTN planning, the planning system formulates a plan by decomposing *tasks* (symbolic representations of activities to be performed) into smaller and smaller subtasks until *primitive* tasks are reached that can be performed directly. The basic idea was developed in the mid-70s (Sacerdoti 1975; Tate 1977). The development of the formal underpinnings came much later, in the mid-1990s (Erol, Hendler, and Nau 1996). HTN planning research has been much more application oriented than most other AI-planning research, and most HTN planning systems have been used in one or more application domains.

An HTN planning problem consists of the following: the *initial state* (a symbolic representation of the state of the world at the time that the plan executor will begin executing its plan), the *initial task network* (a set of tasks to be performed, along with some constraints that must be satisfied), and a *domain description* that contains the following:

(1) A set of *planning operators* that describe various kinds of actions that the plan executor can perform. These are similar to classical planning operators such as the ones in Planning Domain Definition Language (PDDL) (McDermott 1998; Fox and Long 2003).
(2) A set of *methods* that describe various possible ways of decomposing tasks into subtasks. These are the "standard operating procedures" that one would normally use to perform tasks in the domain. Each method may have a set of constraints that must be satisfied to be applicable.

(3) Optionally, various other information such as definitions of auxiliary functions and definitions of axioms for inferring conditions that are not mentioned explicitly in states of the world.

Planning is done by applying methods to non-primitive tasks to compose them into subtasks, and applying operators to primitive tasks to produce actions. If this is done in such a way that all of the constraints are satisfied, then the planner has found a solution plan; otherwise the planner will need to backtrack and try other methods and actions.

In this paper, we use a form of HTN planning called *Ordered Task Decomposition* (Nau et al. 1999) in which, at each point in the planning process, the planner has a totally ordered list of tasks to accomplish and does a forward search to accomplish those tasks. The impact of this assumption is that at each point of the planning process, the state of the world and the next task to be achieved are exactly known. This makes the reasoning of our algorithm easier because at each point in the planning process it knows exactly what the current state of the world is and which methods are (or are not) applicable under this state of the world. We describe this more fully in Section 3.2.

2.2. Candidate Elimination

Candidate elimination is a well-known machine-learning algorithm introduced in (Mitchell 1977). Several extensions of the original algorithm have been proposed (Hirsh 1994; Sebag 1995; Hirsh, Mishra, and Pitt 1997, 2004). Candidate elimination is based on the concept of a *version space*, the set of possible explanations of the concept that is being learned. This concept is represented by two sets: a set G of maximally general possible predicates to explain the concept, and a set S of maximally specific possible such predicates. Every concept between these two borders is a member of the version space, and is a possible explanation of the concept being learned. If enough training samples are given, the version space converges to a single answer (i.e., sets S and G become equivalent). It is also possible that the version space can collapse if the training samples are inconsistent.

3. PROBLEM INPUT AND OUTPUT SPECIFICATION

There are at least two approaches that one might consider for learning preconditions of HTN methods. First, a lazy (e.g., Case-Based Reasoning (CBR)) approach can be used to directly replay plans previously generated by the human expert. It assumes that plans that were successfully used in situations similar to the current situation are likely to work now. Second, an eager approach can be used to induce methods that could be used to mimic human expertise. In this approach, the training samples, which include situations where HTN methods are (or are not) applicable to decompose their corresponding tasks to their subtasks, are generalized to make the learner capable of predicting if the same methods are applicable in the situations not seen before. In either approach, adding new training samples, which represent human expert activities while solving an HTN planning problem, is expected to yield better approximations of the domain. Due to the complexity of the semantics of HTN planning, one should carefully define the inputs and outputs of the learning algorithm and what *learning* means in this context. In this paper, we have chosen an eager approach.

3.1. What Kind of Input to Use

For supervised learning of domains (either in an action-based or an HTN planning environment), two possible forms of input are:

(1) A set of previously generated plans. These plans can be generated in several ways (e.g., by a human expert in that domain). The learning process consists of extracting domain information from these plans.
(2) A collection of *plan traces*, which contain not only the correct solution for a planning problem, but also information about inferences derived and decisions made while this plan was generated (e.g., by a human expert). Note that in both these cases, a correct solution may be "NO PLAN" in situations where there is no plan to solve a given planning problem.

The second form of input is preferable because it will result in faster and more accurate learning; plan traces contain much more information about the domain being learned than plans. For this reason, most previous related work has used the second form of input. For example, in PRODIGY, a system that uses learning techniques in an action-based planning context, *derivational traces* are used in the learning process (Veloso and Carbonell 1993). These traces contain information about the planner's internal right and wrong decisions in addition to the final solution.

In this paper, we also use this second form of input, with appropriate adaptations for use in an HTN planning environment rather than an action-based planning environment.

In addition to the plan traces, we assume that the input to the learning algorithm includes an *incomplete version* of the *domain* (both terms are defined formally in the next subsection). This assumption means that when an HTN method precondition learner is learning preconditions, it knows in advance how to decompose tasks, but it does not known under what preconditions each of the several methods to decompose a task is applicable and should be chosen. This can happen for instance in a military operation, where the overall strategy of dividing tasks to subtasks is usually dictated by the military doctrine and therefore known, but the tactical decision on which on the available methods should be used is made based on the situation on the ground.

The only missing parts in the incomplete version of a domain are the preconditions of the methods in that domain. Therefore, giving the incomplete version of the domain as input to an HTN method precondition learner means that the learner can return a set of complete HTN domains rather than a set of possible method preconditions as its output by adding the learned method preconditions to the given incomplete version of the domain. Note that here we talk about the learner returning a set of complete HTN domains instead of returning just one such domain. This is because the learner may not get enough training data to fully converge, in which case there might be more than one HTN domain description that match the training data.

By definition, the incomplete version includes the (complete and correct) operator definitions, which seems reasonable given that in HTN context operators usually denote concrete and simple actions with obvious effects. For research on how to learn operator preconditions and effects in more general cases, see Gil (1992, 1993, 1994).

3.2. Definitions

In this paper, we use a form of HTN planning called *Ordered Task Decomposition* (Nau, Smith, and Erol 1998) in which, at each point in the planning process, the planner has a totally ordered list of tasks to accomplish. Our definitions are based loosely on the ones in (Nau et al. 1999; Ghallab, Nau, and Traverso 2004).

An *ordered HTN domain* is a triple (T, M, O) where:

(1) T is a list of *tasks*. Each task has a name and zero or more arguments, each of which is a variable symbol.[2] Each task can be either *primitive* or *nonprimitive*. A primitive task represents a concrete action that can be carried out directly, while a nonprimitive task must be decomposed into simpler subtasks. In this paper, primitive tasks are represented by PT or PT_i for some integer i, and nonprimitive tasks are represented by NT or NT_i for some integer i.

(2) M is a collection of *methods*, each of which is a triple $m = (NT, DEC, P)$, where NT is a nonprimitive task, DEC is a totally ordered list of tasks called a *decomposition* of NT, and P (the *precondition*) is a boolean formula of first-order predicate calculus. Every free variable in P that does not appear in the argument list of NT is assumed to be existentially quantified, and every variable in DEC must appear either in the argument list of NT or somewhere in P.

(3) O is a collection of *operators*, each of which is a triple $o = (PT, DEL, ADD)$, where PT is a primitive task, and DEL and ADD are the sets of logical atoms that will be, respectively, deleted from and added to the world state when the operator is executed. All variables in DEL and ADD must appear in the argument list of PT. We also assume that for each primitive task $t \in T$, there is at most one operator (PT, DEL, ADD) such that t unifies with PT (i.e., each primitive task can be performed in at most one way). Unlike classical planning operators, our operators do not have preconditions. However, this does not affect the expressivity of the formalism, because it is easy to map an HTN planning domain in which the operators have preconditions into an equivalent HTN planning domain in which no operators have preconditions. The construction is as follows: if o is an operator that achieves some primitive task PT and has a precondition P, replace o with a method m that has o's precondition and decomposes a new nonprimitive task NT into PT, plus an operator o' that has o's add and delete lists but no precondition. In addition, replace all other occurrences of PT in the domain description with NT.

Figure 1 shows a sample domain with five methods (represented by bigger rectangles) and four operators (represented by circles). This domain is used later in this paper as an example several times. The smaller rectangles show the subtasks into which each method decomposes its corresponding task. The vertical arrows (labeled "Decomposition") correspond methods with their task decompositions. The horizontal arrow shows an ordering constraint on a task decomposition. The precondition of each method and the nonprimitive task it decomposes appear inside the rectangle representing it. The precondition and add and delete list of each operator and the primitive task it performs appear inside the circle representing it. For example, method m_1 decomposes task $NT_1(?x)$ into an ordered list of three subtasks $PT_1(?x)$, $NT_2(?x)$, and $NT_3(?x)$ under the precondition that $a(?x)$ holds in the current state of the world when m_1 is being applied.

An *ordered HTN planning problem* is a triple (I, S, D), where I (the initial task list) is a totally ordered list of ground instances of tasks (i.e., members of T) to be performed, S (the initial state of the world) is a set of *ground* logical atoms, and $D = (T, M, O)$ is an ordered HTN domain. In general, HTN tasks are more general than the goals used in classical planning (Ghallab et al. 2004, Chapter 11).

A *binding* θ is a set of pairs (V, C), where V is a variable symbol and C is a constant symbol. In other words, θ is a substitution that replaces variables with constants. The result of applying a binding θ to an expression e is denoted by $e\theta$.

[2] We denote variable symbols by names that begin with question marks, such as $?x$.

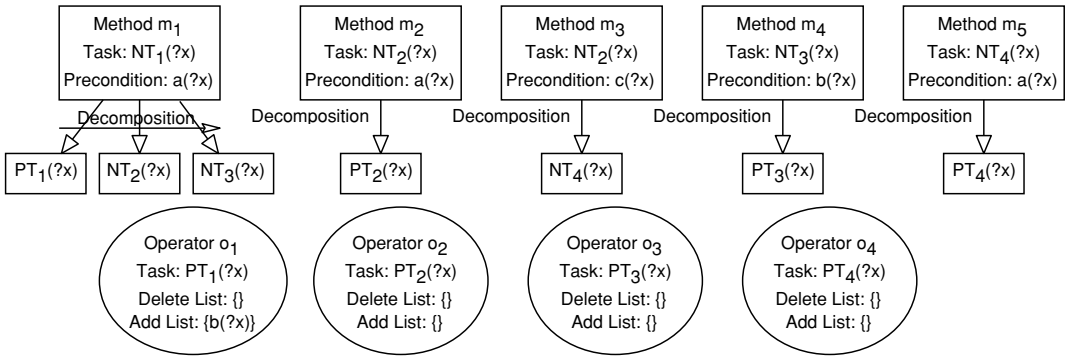

FIGURE 1. A sample domain.

Let $m = (NT, DEC, P)$ be a method. Then $\bar{m} = (NT, DEC)$, i.e., m without its preconditions, is called the *incomplete version* of m. Similarly, if $m\theta = ((NT)\theta, (DEC)\theta, P\theta)$ is an instance of a method m, then the incomplete version of $m\theta$ is $\bar{m}\theta = ((NT)\theta, (DEC)\theta)$.

Assumption 1. We assume that no two different methods $m = (NT, DEC, P)$ and $m' = (NT, DEC, P')$ can exist such that $P \neq P'$. Therefore, there is a one-to-one correspondence between a set of methods $\{m_1, \ldots, m_k\}$ and their incomplete versions $\{\bar{m}_1, \ldots, \bar{m}_k\}$, allowing each incomplete version $\bar{m}_i$ to represent m_i.

Note that if we allow disjunctions in preconditions, Assumption 1 does not affect expressivity because we can replace m and m' with one single method $(NT, DEC, P \vee P')$.

Let $D = (T, M, O)$ be an ordered HTN domain. Then, if $\bar{M}$ denotes the set of incomplete versions of all methods in M, $\bar{D} = (T, \bar{M}, O)$ is defined to be *the incomplete version* of D.

A method (NT, DEC, P) is *applicable* in a state S if there is a binding θ that binds all variables in NT's argument list such that $P\theta$ is satisfied in S.

A *decomposition tree* for a task t in an ordered HTN domain D is an ordered tree ϵ having the following properties:

(1) ϵ's root is t;
(2) each non-leaf node of ϵ is a nonprimitive task and each leaf node is a primitive task;
(3) for each non-leaf node NT of ϵ, there is a method $m = (NT, (C_1, \ldots, C_k), P)$, where $(C_1, \ldots, C_k)$ is the ordered list of the children of NT.

In a decomposition tree ϵ, a node u *occurs before* a node v if:

(1) There is a node w such that u is the ith child of w and v is the jth child of w, where $i < j$.
(2) Or u and v have ancestors u' and v', respectively, such that u' occurs before v'. (Note that we may have $u' = u$ or $v' = v$.)

Given these definitions, it follows that the leaves of a decomposition tree are totally ordered by the "occurs before" relation.

Figure 2 shows all possible decomposition trees for the task NT_1 in Figure 1. The expressions within the brackets in this figure show the preconditions of the methods that are being used in each node to decompose its associated task.

The *prolog* of a node r in a task decomposition tree ϵ is a set of preconditions of all the proper descendants of r in ϵ such that for each such precondition P of such a node n there

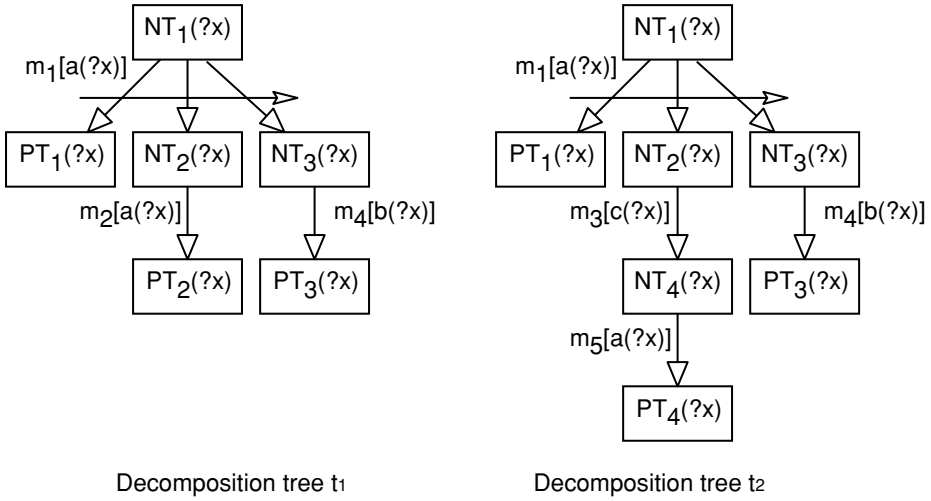

FIGURE 2. All possible decomposition trees for task NT_1 in Figure 1.

is no operator in ϵ that has P in its add list and occurs before n. For any method m to be successfully applicable to decompose the task associated with r to its subtasks, the prolog of r in addition to the precondition of m must be satisfiable in the current state of the world. For example, in Figure 2, in decomposition tree t_1, the prolog of NT_1 is $\{a(?x)\}$, while in decomposition tree t_2 the prolog of NT_1 is $\{a(?x), c(?x)\}$. Note that $b(?x)$ is not in either prolog because it is in the add list of operator o_1, which is the only way to achieve primitive task PT_1. This means that both decomposition trees are applicable to accomplish task NT_1 even if $b(?x)$ does not hold in the state of the world before decomposition of NT_1, because it will be added by operator o_1 to the state anyway. On the other hand, if $c(?x)$ does not hold in the state of the world before decomposing NT_1, the decomposition tree t_2 will not be applicable because there is no operator in it that establishes $c(?x)$ before it is required to decompose NT_2.

A task decomposition tree ϵ is *consistent* if for each node n in ϵ with precondition P and prolog P', $P \wedge P'$ is satisfiable, and either there is no operator o that occurs before n in ϵ that deletes P or P', or if there is such an operator, then there is another operator o' that occurs after o and before n that reasserts the deleted precondition. For example, in Figure 2 both decomposition trees are consistent. However, if we change the precondition of method m_1 from $a(?x)$ to $\neg a(?x)$, they will both become inconsistent (because they will both need $\neg a(?x)$ to apply m_1, and then $a(?x)$ to apply m_2 or m_5). Also, if we add $b(?x)$ to the delete list of operator o_2, the decomposition tree t_1 will become inconsistent (because the precondition of m_4 will be deleted, which makes it impossible to use it to decompose NT_3).

A *partial solution tree* for a ground instance g of a task t in an ordered HTN domain $D = (T, M, O)$ is a substitution instance of a decomposition tree for t, i.e., it is an ordered tree τ having the following properties:

(1) τ's root is g;
(2) each node of τ is a ground instance of a task;
(3) for each non-leaf node N of τ, there is a method $m = (NT, DEC, P)$ and a binding θ such that $N = (NT)\theta$ and $(C_1, \ldots, C_k) = (DEC)\theta$, where $(C_1, \ldots, C_k)$ is the list of the children of N. $m\theta = ((NT)\theta, (DEC)\theta, P\theta)$ is called the *method instance* for N.

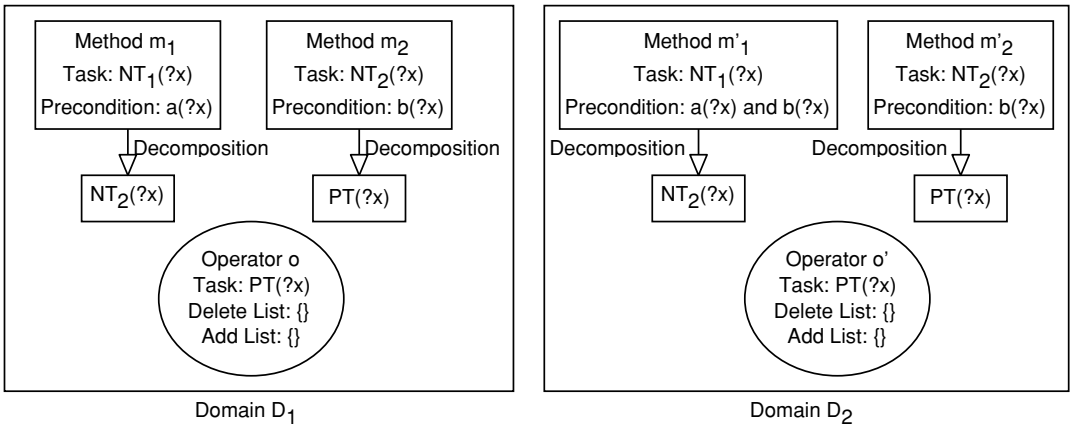

FIGURE 3. Domains D_1 and D_2 are equivalent.

A *partial solution forest* for an ordered HTN planning problem (I, S, D) is a totally ordered list $(\tau_1, \ldots, \tau_k)$ of partial solution trees, one for each member of I.

In a partial solution forest $F = (\tau_1, \ldots, \tau_k)$, a node u *occurs before* a node v in each of the following cases:

(1) u and v are in different trees τ_i and τ_j, respectively, where $i < j$.
(2) There is a node w such that u is the ith child of w and v is the jth child of w, where $i < j$.
(3) u and v have ancestors u' and v', respectively, such that u' occurs before v'. (Note that we may have $u' = u$ or $v' = v$.)

Given these definitions, it follows that the leaves of a partial solution forest are totally ordered by the "occurs before" relation.

A *solution* for an ordered HTN planning problem (I, S, D) is a partial solution forest F for (I, S, D) having the following properties:

(1) For every leaf node L of F, there is an operator $o = (PT, DEL, ADD)$ and a binding θ such that $L = (PT)\theta$. $o\theta$ is called the *operator instance* for L.
(2) For every non-leaf node N of F, let $m = (NT, DEC, P)$ be the method instance for N. Let $(L_1, \ldots, L_k)$ be the totally ordered list of leaf nodes that occur before N, and let $o_1, \ldots, o_k$, respectively, be the operator instances for these nodes. Then P is satisfied in the state produced by starting with S and applying $o_1, \ldots, o_k$ sequentially.

Two ordered HTN domains D_1 and D_2 are *equivalent* if and only if, for any arbitrary task list I and state of the world S, the ordered HTN planning problems (I, S, D_1) and (I, S, D_2) have exactly the same set of possible solution forests. For example, domains D_1 and D_2 in Figure 3 are equivalent.

For a method $m = (NT, DEC, P)$, a precondition is *redundant* if for each possible task decomposition tree of NT that uses m to decompose its root, that precondition appears in the prolog of NT. For instance, in Figure 3, precondition $b(?x)$ is redundant for method m'_1 in domain D_2. As a more complicated example, precondition $a(?x)$ in method m_1 in Figure 1 is redundant.

A method (NT, DEC, P) is *active* if there exists at least one consistent task decomposition tree with root NT that uses this method to decompose its root. In other words, if a method is *dead* (i.e., not active), it cannot possibly be used in a solution to any arbitrary ordered HTN planning problem. For example, in domain D_1 in Figure 3, both methods are active, but if we change the precondition of method m_1 from $a(?x)$ to $\neg b(?x)$, then m_1 will become dead (although m_2 will still be active).

The *canonical form* of a domain D is derived by eliminating first all the dead methods and then all redundant preconditions in D. Obviously, each domain is equivalent to its canonical form.

We are now ready to formally define plan traces and our inputs. A *plan trace* Π consists of (1) a solution to an ordered HTN planning problem and (2) for each internal node N in the solution forest, an incomplete version of all instances of methods that were applicable. This will obviously include the instance of the method that was actually used to decompose N.

The *inputs* for an HTN method precondition learning algorithm consist of the following:

(1) The incomplete version $\bar{D} = (T, \bar{M}, O)$ of a domain $D = (T, M, O)$.
(2) A set of n ordered HTN planning problems $\{(I_j, S_j, D)\}$, where $1 \leq j \leq n$.
(3) A plan trace Π_j for each of these problems.

One of the challenges in any learning algorithm is how to define a criterion to evaluate its output. Often, there is not enough information in the input or there are not enough training samples to derive an optimal output. Therefore, learning algorithms may return a set of *candidate* answers instead of a single answer. This phenomenon can affect the definition of soundness and completeness, which play a crucial role in evaluating outputs in a planning context.

Before we can formally present soundness and completeness definitions in the context of HTN method precondition learning, we need to define *consistent* answers.

A domain D_1 is *consistent with* another domain D_2 with respect to a set of pairs (I_j, S_j) of a task list I_j and a state of the world S_j if and only if, for every j, the plan traces for the ordered HTN planning problems (I_j, S_j, D_1) and (I_j, S_j, D_2) are exactly the same. This definition says that although there may be differences in the methods and/or operators in D_1 and D_2 when solving problems relative to the task list I_j and state S_j for each j, the resulting traces are identical. For example, in Figure 4, domains D_1, D_2, and D_3 are consistent with respect to set $\{(I_1, S_1), (I_2, S_2)\}$, although they are different domains. In domain D_1, the precondition of the method m_1 indicates that there should be an atom $a(?x)$ in the current state of the world, where $?x$ is the argument of the task, in domain D_2, the precondition of the method m_2 indicates that there should be an atom $a(?y)$ in the current state of the world where $?y$ can be any constant symbol (free variables are assumed to be existentially quantified), and finally in domain D_3, the precondition of the method m_3 indicates that there should be either atom $a(c)$ or atom $a(d)$ in the current state of the world. On the other hand, with respect to set $\{(I_1, S_1), (I_1, S_2)\}$, only domains D_2 and D_3 are consistent.

Note that the notion of two domains being equivalent is a specific form of two domains being consistent with each other. Two domains are equivalent if and only if they are consistent with each other with respect to any set of possible pairs of task lists and initial states of the world.

Using the above definition, we can now formally define soundness and completeness for HTN method precondition learning algorithms.

Consider an HTN method precondition learning algorithm whose inputs are

(1) $\bar{D} = (T, \bar{M}, O)$, the incomplete version of a domain $D = (T, M, O)$;

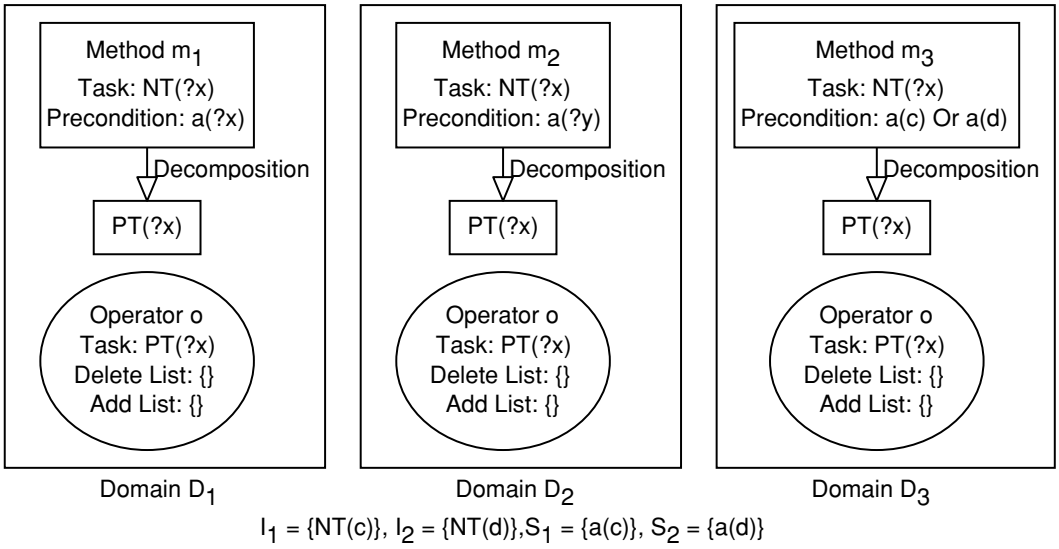

FIGURE 4. All these domains are consistent with respect to set $\{(I_1, S_1), (I_2, S_2)\}$. Domains D_2 and D_3 are consistent with respect to set $\{(I_1, S_1), (I_1, S_2)\}$.

(2) a set of ordered HTN planning problems $\{(I_j, S_j, D)\}$, where $1 \leq j \leq n$;
(3) a plan trace Π_j for each of these ordered HTN planning problems.

An HTN precondition learning algorithm is *sound* if and only if whenever it returns a set of ordered HTN domains, each of them is consistent with D with respect to the set of all (I_j, S_j) pairs.

An HTN method precondition learning algorithm is *complete* if and only if for every domain D' that is consistent with D with respect to the set of all (I_j, S_j) pairs, the algorithm's answer includes a domain that is equivalent to D'.

4. CAMEL: AN HTN PRECONDITION LEARNER BASED ON CANDIDATE ELIMINATION

4.1. Motivation

The main goal of learning HTN methods is to be able to generate plans for new planning problems (or *queries*). This ability will be obtained by learning how to plan using a set of previously generated plans or plan traces. In the machine learning literature, two entirely different kinds of learning, namely *lazy learning* and *eager learning* are discussed. In the purest form of lazy learning, training samples are simply stored. At query time, the algorithm compares the query instance with its recorded training samples. Thus, learning time is minimized while query time can be high, especially if no attempt is made to prune the number of stored training samples. On the other hand, eager learners during the training process generalize the training data into an abstract concept that matches the training data well in the hope that this abstract concept will work well on the cases not included in the training data too. At query time, this concept, rather than the training samples themselves,

are used to answer the query. Thus, learning time is higher than for purely lazy algorithms, while query time is usually lower.

In the context of learning method preconditions, lazy learning has been done using CBR, which involves locating those training samples that are most similar to the planning problem given as the query (Hanney and Keane 1996; Veloso, Muñonz-Avila, and Bergmann 1996; Lenz et al. 1998). This problem is then solved by adapting the solutions stored in the retrieved training samples. Our focus in this paper is on eager learning. Some of the advantages of using eager learning in our context are

(1) *Less query time:* Lazy learning is useful when the number of training samples and frequency of query arrival is small. However, when these numbers are large, finding the most similar training samples can be time consuming (i.e., assuming that no smart indexing method is used). In such situations, eager learners are preferable.
(2) *Reduced knowledge base size:* Once the methods are learned, the system can discard the training samples and use the induced methods to solve new planning problems. In other words, the learned methods will act as a compact *summary* of the training cases. In contrast, purely lazy approaches require that, for every new planning problem, all previously seen examples must be revisited and, therefore, must be stored in the algorithm's knowledge base. Also, although several algorithms exist for significantly reducing storage requirements for case-based classifiers, they do not yet exist for case-based HTN planners.
(3) *Easier plan generation:* If we learn methods completely, then the process of generating a plan for a new planning problem will be much easier. This requires inducing methods for a hierarchical planner so that it can automatically generate plans for new planning problems. In contrast, a case-based planner must dynamically decide, for each new problem, which stored case or combination of cases to apply.

As mentioned earlier, HTN method precondition learning algorithms may not be given enough information in the training set to derive a single exact domain as output. Therefore, the algorithm may return a set of domains. In these situations, a policy is needed to decide which possible domains should be output by the algorithm. Two extreme policies are

(1) The *minimal* policy: Any possible domain is added to the output set if and only if there is enough evidence in the input to prove that this domain *must* be in the output.
(2) The *maximal* policy: Any possible domain is added to the output set if and only if there is not enough evidence in the input to prove that this domain *must not* be in the output.

The minimal policy yields a sound algorithm while the maximal policy yields a complete algorithm. However, both of these extreme approaches perform poorly and neither is both sound and complete. This is because there may be possible domains whose existence in the output set cannot be proved or disproved using the current input, simply because there is not enough information in the input to do so and more training samples are required. These possible domains are discarded in the minimal view and added to the output set in the maximal view.

One way to obtain better performance is to track the possible domains that cannot be proved or disproved so that they can be assessed in the future, after more training samples are acquired. Thus, we need an algorithm for tracking possible domains, while preferably maintaining its soundness and completeness.

Given all this, using candidate elimination seems like a natural choice. It has two main advantages. First, the resulting algorithm is sound and complete if our output is the set of

all possible domains, where each method's set of preconditions can be any member of its corresponding version space. Secondly, candidate elimination is an *incremental* algorithm: whenever the algorithm acquires a new training sample, it just updates its version spaces and discards that training sample afterwards. There is no need to keep the training samples.

Candidate elimination is indeed a very general algorithm. However, to apply this algorithm for a specific application, a generalization topology (a lattice) on the set of possible concepts must be defined (i.e., the generalization/specialization relation, along with the top and bottom of the lattice). In our algorithm, CaMel, every method has a corresponding version space, and each member of a method's version space is a possible precondition for that method.

Candidate elimination requires both negative and positive examples. The concept of a "negative" example in a planning context may not be clear. However, for our context, negative examples can be generated easily. In our definitions, the input to a precondition learner includes plan traces, each of which lists all applicable method instances that can be used to decompose a task instance in a specific world state. Therefore, if there are methods in the domain to decompose the same task that are not listed, we can infer that they were *not* applicable in those specific world states, and can hence serve as negative examples.

To make candidate elimination useful in an HTN planning context, we need to make significant extensions to it. These are described in the next section.

4.2. Algorithm

Before detailing with CaMeL, we need to introduce the notion of *normalization*. Suppose that an HTN precondition learner is trying to learn the preconditions of a method m that is used to decompose the task $move(?t, ?s, ?d)$. Assume that two instances of m are given. One instance is used to decompose the task $m_1 = move(truck_1, city_1, city_2)$ in the state $S_1 = \{truck(truck_1), at(truck_1, city_1)\}$, while the other instance is used to decompose the task $m_2 = move(truck_2, city_3, city_4)$ in the state $S_2 = \{truck(truck_2), at(truck_2, city_3)\}$. Apparently, these two training samples contain the same piece of information. Intuitively, it is "You can move an object $?t$ from any starting location $?s$ to any destination $?d$ if $?t$ is a truck and it is initially at $?s$." However, how is a learning algorithm supposed to derive such a general statement from such a specific example? This statement contains three variables, while the facts are about specific constants such as $truck_1$, $city_1$, and $city_2$. A generalization process is required that changes these constants to variables. This is roughly what happens during a normalization process.

Assumption 2. No two variables in the parameter list of any method m in the domain are mapped to the same constant when m is instantiated in a plan trace.

Assumption 2 is not true in all HTN planning domains, but it is true in many of them, including the ones in our experiments.[3] In domains where this assumption is satisfied θ replaces each variable V_i with a different constant C_i; hence θ has an inverse θ^{-1} which replaces every constant C_i with its corresponding variable V_i.

Consider a ground instance $m\theta$ of a method m that decomposes task t for world state S. Then, $S\theta^{-1}$ is called a *normalization* of S with respect to $m\theta$, and θ^{-1} is called a *normalizer* for S. For the above example, both the normalization of S_1 with respect to m_1 and the normalization of S_2 with respect to m_2 yield $\{truck(?t), at(?t, ?s)\}$.

[3] Formally, CaMeL can still be used even in domains that do not satisfy Assumption 2, because there exists a mapping that transforms any domain that does not satisfy Assumption 2 into an equivalent domain that satisfies Assumption 2. However, the mapping is not entirely satisfactory, because the new domain can be combinatorially larger than the old one.

The normalization process replaces constants in different training examples that play the same role (e.g., $truck_1$ and $truck_2$) with a variable (e.g., $?t$) to generalize the facts that are given as input. Fact generalization is indeed a basic strategy in most eager learning algorithms.

Two kinds of constants may appear in an HTN plan trace Π or its corresponding ordered HTN planning problem (I, S, D). First, *explicit* constants appear explicitly in the domain definition D (i.e., in the effects of operators, or preconditions of methods). Second, *implicit* constants are those that do not appear in the D explicitly. These constants appear in a plan trace because some of the variables in D were instantiated to them while the plan trace was created.

For example, consider an HTN method m for decomposing the task $go(?x, ?y)$ that decomposes this task to a primitive task $walk(?x, ?y)$. The precondition of this method is $weather(good) \wedge at(?x)$. Now, if there are two atoms $at(home)$ and $weather(good)$ in the current world state and an instance of the method m is used to decompose the task $go(?x, ?y)$ to the subtask $walk(home, station)$ in the corresponding plan trace, then the constants *home* and *station* are implicit constants that appear in problem definition (i.e., in I or S) rather than in domain definition, while *good* is an explicit constant, because it appears in D.

As another example, in the blocks world domain, *table* is an explicit constant while names of blocks are implicit constants.

When normalization is used to generalize the training samples, the following crucial assumption is made.

Assumption 3. In an ordered HTN planning problem (I, S, D) an explicit constant cannot appear anywhere in I. Also, an explicit constant can appear in an atom a in S only if it plays the same role in a and in the predicate in D in which it appears. That is, it should appear only in the same predicate and in the same position in that predicate. For instance, the constant *good* in the above example cannot appear anywhere in I or S other than in atom $weather(good)$.

Assumption 3 is needed to avoid the situation in which a variable in a precondition is instantiated to a constant, which coincidentally is the same as another constant somewhere else in the precondition. It makes sure that the normalization process will never result in overgeneralized predicates.

For an example showing why normalization is useful and why Assumption 3 should hold to use normalization, see Figure 4. Assume that an HTN method precondition learner is trying to find the most specific precondition for the method in that figure and it is given $\{(I_1, S_1), (I_2, S_2)\}$ as the training set. If the learner does not make Assumption 3, it will not have any way to choose between methods m_1 and m_3. This is because it will have no way to tell if the appearance of c or d in both the task list to be achieved and the current state of the world ($\{(I_1, S_1)\}$ and $\{(I_2, S_2)\}$, respectively) is just a mere coincidence, or it happens because there is a variable in the precondition that instantiates to c or d as a parameter of the task NT. However, given Assumption 3 and using normalization, the learner can deduce that c and d are not explicit variables (because they appear in I_1 and I_2, respectively), which means a variable in the argument list of NT must have been instantiated to them, which means in turn that m_1, and not m_3, is the method the learner is looking for.

Theorem 1. Assumption 3 does not limit the expressivity of the HTN domains CaMeL can handle.

Proof. There is a simple algorithm to transform a domain D where Assumption 3 does not hold to an equivalent domain where it does. For each explicit constant c in a planning problem (I, S, D) that occurs in I or in a wrong place in S (i.e., in a predicate other than the predicate in D in which c appears, or in a different place in the same predicate) , create a new predicate symbol $pred_c$, and add the atom $pred_c(c)$ to S. In the domain, do the following

Given:

$\bar{D} = (T, \bar{M}, O)$, the incomplete version of $D = (T, M, O)$

$I = \{I_1, \ldots, I_n\}$, a set of task lists

$S = \{S_1, \ldots, S_n\}$, a set of world states

$\Pi = \{\Pi_1, \ldots, \Pi_n\}$, a set of plan traces, one per ordered HTN planning problem (I_i, S_i, D)

CaMeL($\bar{D}$,I,S,Π)

  C = GetConstants(I)

  FOR each method $\bar{m}_j \in \bar{M}$

    Initialize a version space VS_j

  FOR each plan trace $\Pi_i \in \Pi$

    FOR each method $\bar{m}_j \in \bar{M}$

      $inst$ = LocateInstances(Π_i,$\bar{m}_j$)

      FOR each (loc,θ,np)$\in inst$

        S'_i = ComputeState($\bar{D}$,S_i,Π_i,loc)

        VS_j = CE(ExtractRelevant(Normalize(S'_i,θ),C),VS_j,np)

        RemoveDeadMethods(D, VS_1,...,VS_m)

        RemoveRedundantPreconditions(D, VS_1,...,VS_m)

        IF Converged(VS_1,...,VS_m)

          RETURN all VS_j's

  RETURN all VS_j's

FIGURE 5. The CaMeL method-learning algorithm.

in every method and operator where c occurs: replace all occurrences of c with a variable symbol $?x_c$, and insert a new precondition $pred_c(?x_c)$ as the first precondition of the method or operator. Thus c no longer appears in the domain—but during problem-solving, $?x_c$ will get bound to c so that c gets used in the same places where it was used before. ■

The pseudocode of our algorithm, CaMeL, is given in Figure 5. The algorithm subroutines are as follows:

(1) GetConstants(I), where I is a task list, returns all the constants that appear somewhere in I. These are the constants that CaMeL knows for sure cannot be explicit constants.
(2) LocateInstances(Π, $\bar{m}$), where Π is a plan trace and $\bar{m} = (NT, DEC)$ is an incomplete method, is a function that returns a set of triples (loc, θ, np). Each of these triples corresponds to one of the places where an instance $(NT)\theta$ of a nonprimitive task NT was decomposed in Π. loc denotes the number of operators that occur before this instance

of a task in the plan trace Π, and *np* is a boolean variable indicating whether method instance $m\theta$ was applicable to decompose $(NT)\theta$.

(3) ComputeState($\bar{D}$, S, Π, loc) is a function that computes the world state after applying the first *loc* operators of the plan trace Π in incomplete-ordered HTN domain $\bar{D}$, where the initial world state is S.

(4) ExtractRelevant(S, C), where S is a normalized state and C is a set of constants that are known not to be explicit. This function returns only predicates in the given normalized state that can potentially be relevant to the particular instance of a method currently being considered. A predicate is relevant if at least one of its arguments is either an explicit constant or a variable symbol (the latter case happens when the constant symbol this variable replaced during the normalization appeared somewhere in the method instance currently being considered), or one of its arguments appears in another potentially relevant predicate's argument list. Therefore, to calculate the potentially relevant predicates, this function starts by initializing its output to the set of facts in S that have at least one variable or one *potential* explicit constant (i.e., not in C) in their argument list. Then in each following iteration, it adds those remaining predicates in S that share at least one constant in their argument list with some other fact already in its output. These iterations continue until no new predicates are added to the output. The facts that are left out after this can not be relevant to the method instance being processed and therefore are not used to update the corresponding version space.

(5) CE(S, VS, *np*) is an implementation of the candidate elimination algorithm on version space VS with training sample S. *np* is a boolean variable that indicates whether S is a negative or positive example.

(6) RemoveDeadMethods(D, $VS_1, \ldots, VS_m$) removes methods that are dead (i.e., methods that can never be used to solve any arbitrary planning problem because some of the preconditions of the nodes in the decomposition tree produced by their application can not be satisfied). Note that because each method's version space can potentially represent more than one candidate for what that method's precondition can be, this function should try all the possible combinations of related methods before labeling a method as dead.

(7) RemoveRedundantPreconditions(D, $VS_1, \ldots, VS_m$) removes all the redundant preconditions of methods. Again, because each method's version space can potentially represent more than one candidate for what that method's precondition can be, this function should try all the possible combinations of related methods before labeling a precondition as redundant.

(8) Converged($VS_1, \ldots, VS_m$) checks if all of the version spaces $VS_1, \ldots, VS_m$ have converged.

The algorithm starts by initializing one version space for each method in the domain being learned. Then, for each plan trace, and for each method, all the instances of that method in that plan trace are found. These include both positive (i.e., method was applicable to decompose its associated task) and negative (i.e., method was not applicable to decompose its associated task) instances. Then, the state of the world at the point each method instance appears is calculated by applying the operators in the plan trace that occur before that instance to the initial state of the world. The resulting state is then normalized according to the binding used in the method instance, and the potentially relevant facts are extracted from each normalized state. Then, each resulting state (which is a set of predicates) is used to update the corresponding version space. After updating all the version spaces, the dead methods and redundant preconditions are removed in that order (the order is important because the

removal of dead methods can produce redundant preconditions). This is repeated till either all the training samples are used, or each of the version spaces converges to a single node.

5. ALGORITHM SOUNDNESS, COMPLETENESS, AND CONVERGENCE

In this section, we discuss more assumptions and restrictions a domain must satisfy in order for CaMeL to work correctly. We will also propose a few theorems about CaMeL in the framework we have defined.

Another assumption should be made because we use the candidate elimination algorithm. In order for candidate elimination to work properly, the terms *the most specific*, *the most general*, *the most specific generalization* and *the most general specification* must be defined for the set of possible members of the version space. CaMeL uses version spaces to show the possible preconditions of methods, which in general can be any first order predicate calculus formula. Unfortunately, the aforementioned terms cannot be defined for the set of all possible boolean formulas in first order predicate calculus.

Assumption 4. Preconditions of the methods have a *known* form, and this form is such that the set of all possible preconditions is partially ordered by the "more general" relation, and the most general and the most specific preconditions exist in this set. Furthermore, for any given precondition P other than the most general preconditions, there exist a nonempty, finite set G of preconditions that are all more general than P, but there is no other precondition that is more general than P but more specific than any of the members of G. Also, for any given precondition P other than the most specific preconditions, there exist a nonempty, finite set S of preconditions that are all more specific than P, but there is no other precondition that is more specific than P but more general than any of the members of S.

Assumptions 3 and 4 together define the learning algorithm's *representational bias*. A representational bias defines the states in the search space of a learning algorithm (Gordon and Desjardins 1995). It guarantees that we can generalize given facts about the training samples (Mitchell 1980).

CaMeL accepts conjunctions, existential quantifiers, negations, and limited disjunctions (meaning that the number of disjuncts in a disjunction is limited to a predefined constant) in method preconditions. The reason we have limited the number of disjuncts is that we wanted to enable CaMeL to learn existential quantifiers: For example, if in a domain any truck can be used to accomplish a method that moves a package from some location to another location, CaMeL might get several plan traces where several different trucks $t_1, t_2, \ldots, t_n$ are used to move the package. If we did not put a restriction on the number of disjuncts in a precondition, CaMeL would have no way to distinguish between (1) a precondition consisting of n disjuncts, each of which uses one of the specific t_is to move the package, and (2) another precondition that simply uses *any* available truck to do so. In our implementation, we limited the number of disjuncts to three, so in this example as soon as a fourth truck is used to move a package, CaMeL will automatically induce that the method precondition has an existential quantifier over the trucks and any truck would do here. Note that there is a trade-off here between how expressive one wants one's domains to be, and how fast one wants CaMeL to converge, so this limit on the number of disjuncts should be chosen carefully.

Given the assumptions we have made so far, the following theorems can be proved.

Theorem 2. CaMeL is a sound HTN precondition learner.

Sketch of Proof. The proof for this theorem is straightforward. It is based on the correctness of the Candidate Elimination algorithm. Assume that CaMeL is not sound. It means that

while learning an ordered HTN domain D, at least one of the domains returned by it, D', is not consistent with D with respect to the training set. That means D' in at least one decision point in one of the plan traces associated with the training set has a different view of which methods are or are not applicable in a given state of the world. Either D' thinks a method is applicable that D thinks is not, or D thinks a method is applicable that D' thinks is not. In the former case, there is a node in the version space of that method that is not consistent with the training data, and in the latter case there is a node consistent with the training data that is not in the version space. Both of these cases contradict the correctness of the version space algorithm. Therefore, CaMeL is a sound HTN precondition learner. ■

Theorem 3. CaMeL is a complete HTN precondition learner.

Sketch of Proof. The proof for this theorem is based on the correctness of the Candidate Elimination algorithm too. Assume that CaMeL is not complete. It means that while learning an ordered HTN domain D, there was at least one domain, D', that was consistent with D with respect to the training set, but was not returned by CaMeL. This means preconditions of at least one of the methods of D' was not in its corresponding version space, which contradicts the correctness of Candidate Elimination algorithm, because this precondition is consistent with this version space given this training set, and therefore it should be in the version space. Therefore, CaMeL is a complete HTN precondition learner. ■

Theorem 4. For any given ordered HTN domain D that satisfies our restrictions and assumptions, there exist a finite set of plan traces that cause CaMeL to converge to the canonical form of D.

Sketch of Proof. To prove this theorem, two lemmas are needed first:

Lemma 1. For each active method, there exist plan traces that use that method.

Lemma 2. For each nonredundant precondition P, there exist plan traces that depend on P, that is, if P is deleted, there will be either inapplicable methods that become applicable (in cases where deletion of P makes the overall precondition less restrict, such as cases where P is a conjunct in a conjunction), or applicable methods that become inapplicable (in cases where deletion of P makes the overall precondition more restrictive, such as cases where P is a disjunct in a disjunction).

To create a finite set of plan traces that cause CaMeL to converge to the canonical form of a domain D that it is learning, consider all the active methods in D. For each such method $m = (NT, DEC, P)$, we can create planning problems of the form (NT, S_i, D). Plan traces associated with these planning problems fall into two categories, according to each S_i. Either m is applicable to decompose NT (such plan traces exist according to Lemma 1), or it is not. These two categories of plan traces can be used as positive and negative examples to learn what P is in the version space associated with this method. According to the position of P in the finite lattice of all possible preconditions (this lattice exists because of Assumption 4), we can set the values of S_is such that as a result of these positive and negative examples, the version space associated with m converges to P. This is possible because according to Lemma 2, all the different nonredundant elements in P can be verified by some plan traces.■

6. EMPIRICAL EVALUATION

Theorem 4 says that there always exists a set of training samples that causes CaMeL to converge if the domain definition satisfies our restrictions and assumptions. However, this

theorem does not give us any information about the number of training samples in such a set. What we need in practice is an answer to the question "How many samples will be needed to converge on average?" In this section, we discuss our experiments to answer this question.

6.1. Test Domains

The first domain we used in our experiments was the well-known blocks world domain. We used an HTN domain description based on the block-stacking algorithm discussed in (Gupta and Nau 1992). Our HTN implementation has six operators and 11 methods.

The second domain we used was a simplified and abstracted version of Noncombatant Evacuation Operation (NEO) planning (Muñoz-Avila et al. 1999). Our HTN implementation has four operators and 17 methods.

6.2. Simulating a Human Expert

One goal in our work is to learn methods for HTN planners in NEO planning domains. These domains are usually complicated, requiring many samples to learn each method. It is difficult to obtain these training samples for these kinds of domains. Even if we had access to the real-world NEO training samples, those samples would need to be classified by human experts and the concepts learned by CaMeL would need to be tested by human experts to assess their correctness. This would be very expensive and time consuming.

To overcome this problem, we decided to *simulate* a human expert. We used a correct hierarchical planner to generate planning traces for random planning problems on an ordered HTN domain. Then we fed these plan traces to CaMeL and observed its behavior until it converged to the set of methods used to generate these plan traces.

The hierarchical planner we used is a slightly modified version of SHOP (Nau et al. 1999). In SHOP, if more than one method is applicable in some situation, the method that appears first in the SHOP knowledge base is *always* chosen. Because in our framework there is no ordering on the set of methods, we changed this behavior so that SHOP chooses one of the applicable methods randomly at each point. We also changed the output of SHOP from a simple plan to a plan trace.

6.3. Generating the Training Set

For blocks world, we generated four sets of random problems with 100, 200, 300, and 400 blocks. We generated planning problems block by block, putting each new block randomly and uniformly onto an existing clear block or the table.

To generate plan traces in the NEO domain, we had to generate a random NEO planning problem and feed it to the modified version of SHOP. To generate a random NEO planning problem, every possible state atom was assigned a random variable, indicating whether or not it should be present in the initial state of the world (e.g., whether there should be an airport in a specific city), or what value its corresponding state atom should have (whether the evacuation operation is to take place in a *hostile*, *neutral*, or *permissive* environment). In our preliminary experiments, we noted that the values taken by most variables did not affect the results in a significant way (i.e., these different values did not result in a noticeable change in either the number of plan traces needed to converge or the learning time). The one exception was the variable indicating if there is an airport in a city. Therefore, we decided to assign a uniform distribution to all random variables other than this variable, and to perform experiments with several different values of this variable, to which we will refer as P in this Section. We conducted 11 sets of experiments, with $P = \frac{1}{12}, \frac{2}{12}, \ldots, \frac{11}{12}$.

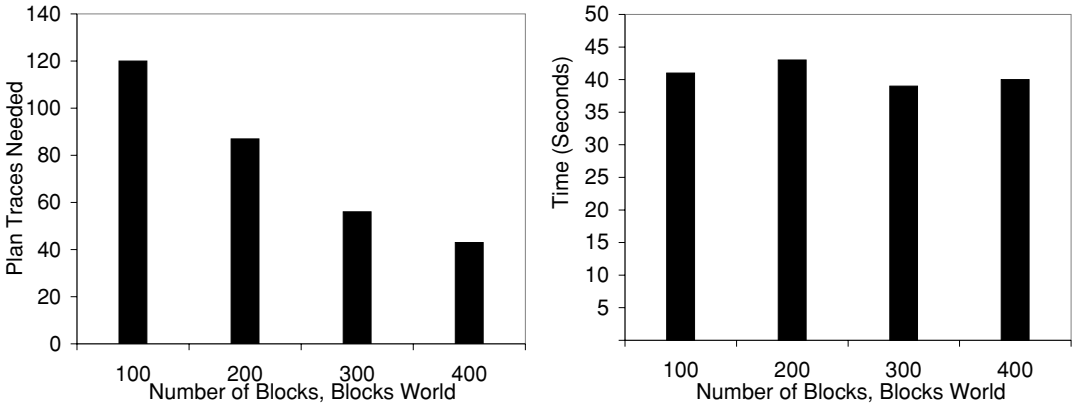

FIGURE 6. Blocks World: Number of plan traces needed to converge (left), CPU time used by CaMeL to converge (right).

6.4. Results

Each of the results in this section is calculated by averaging the results in 10 different sets of randomly generated training sets. There was little variation in the results of each individual run: the difference between each such result and the average of the results (reported here) was never more than 10%.

Figure 6 shows the number of plan traces needed to converge in the blocks world and the CPU time used by CaMeL to do so.[4] As expected, the number of plan traces needed by CaMeL to converge decreases as larger problems are given. The reason is that more information is provided in each plan trace. However, the learning time seems to be roughly the same for different problem sizes: the extra time needed to handle larger problem sizes offsets the facts that CaMeL needs fewer plan traces.

For the NEO domain, after generating the 11 sets of training samples for $P = \frac{1}{12}, \frac{2}{12}, \ldots, \frac{11}{12}$, we fed each training set to CaMeL until it converged. Figure 7 shows the number of plan traces needed in each case in order for CaMeL to converge, and the time in seconds CaMeL needed to converge in each of those cases. As can be seen, the number of required plan traces and time is minimized when the probability P of a city having an airport is approximately 50% (i.e., the sixth training set). For other values of P, methods are harder to learn. When P is close to 0 (i.e., first training set), the hard-to-learn methods are those whose preconditions require cities to have airports, because the cases where these methods are applicable somewhere in given plan traces are so rare that the learner cannot easily induce their preconditions. When P is close to 1, the preconditions that are hard to learn are those methods that do not require cities to have airports. the probability that there is an airport whenever these methods are applicable is high. As a result, the learner cannot induce that an airport's presence is not required.

In our experiments, CaMeL learned some methods much more quickly than others. The first graph in Figure 8 shows how many of the methods are learned completely as a function of the number of plan traces, for the cases where $P = \frac{3}{12}, \frac{5}{12}, \frac{7}{12}, \frac{9}{12}$. From examining the raw data that went into this figure, we observe the following:

[4]These experiments were conducted on a Sun Ultra 10 machine with a 440 MHz SUNW UltraSPARC-IIi CPU and 128 megabytes of RAM.

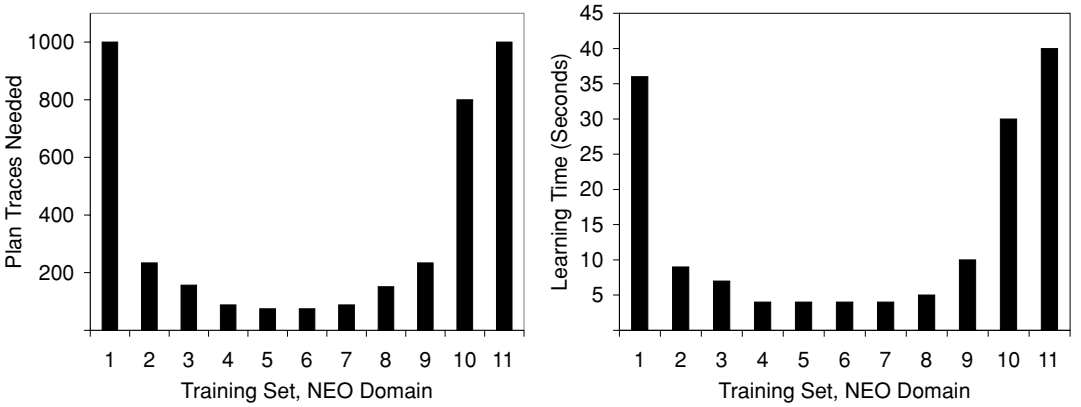

FIGURE 7. NEO Domain: Number of plan traces needed to converge (left), CPU time used by CaMeL to converge (right)

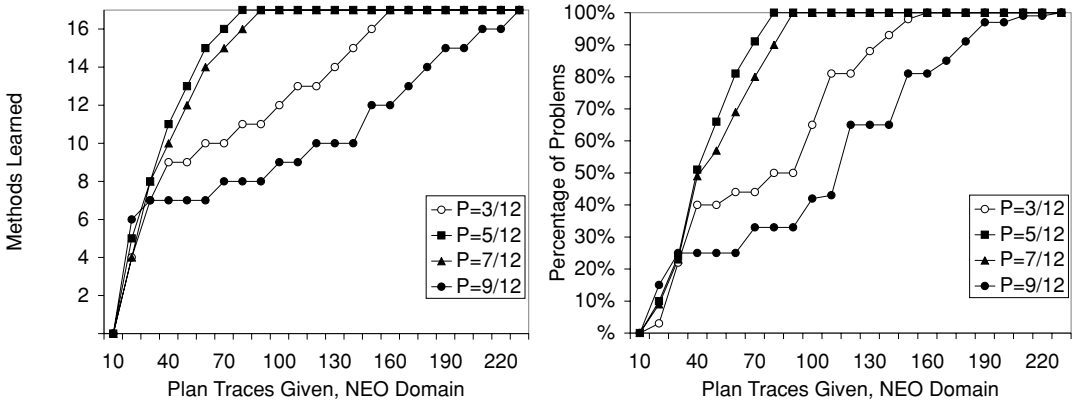

FIGURE 8. Speed of convergence for different training sets in NEO domain, and percentage of problems SHOP could solve using the methods that had already converged. The total number of methods in the domain is 17.

(1) When P is close to 50%, all methods are learned very quickly.
(2) When P is close to 0, methods whose preconditions do not require cities to have airports are learned very quickly.
(3) When P is close to 1, methods whose preconditions require cities to have airports are learned very quickly.

We believe that these observations are quite important. When P is close to zero, methods that do not use airports are more likely to be used to decompose the tasks in the training set, which makes them easier to learn. If we believe that the planning problems in the test set follow the same distribution as the planning problems in the training set, it can be argued that these quickly-learned methods are the ones that will be the most useful and therefore the most important to solve the planning problems in the test set. The same argument can be made when P is close to 1 with the methods that require cities to have airports. In other words,

CaMeL learns the most useful methods quickly, suggesting that CaMeL may potentially be of use in real world domains even if only a small number of training samples are available.

To test the hypothesis that CaMeL learns most of the methods quickly, we conducted another experiment. In this experiment, CaMeL tries to solve problems in the NEO domain even before complete convergence is achieved. To solve the planning problems, CaMeL used only methods whose preconditions were learned completely, omitting all other methods from the domain description. The results of this experiment for the cases where $P = \frac{3}{12}, \frac{5}{12}, \frac{7}{12}, \frac{9}{12}$ are shown in the second graph in Figure 8. As can be seen, in the cases where learning was slower, i.e., the cases where $P = \frac{3}{12}$ or $\frac{9}{12}$, almost four-fifth and two-third of problems can be solved respectively using less than 120 plan traces, although with 120 plan traces there are, respectively, 4 and 8 methods out of a total of 17 yet to be completely learned, and learning the entire domain requires more than 200 plan traces in each case (see first graph in Figure 8). This suggests that more useful methods have already been learned and the extra plan traces are needed only to learn methods that are not as common. It also suggests that CaMeL can be useful in solving a lot of problems long before all methods are learned.

7. RELATED WORK

Much of the work done on the integration of learning and planning is focused on conventional action-based planners, rather than HTN planning as in CaMeL. Usually, this work, as formulated in Minton (1990), is aimed at speeding up the plan generation process or to increase the quality of the generated plans by learning search control rules. These rules give the planner knowledge to help it decide at choice points and include *selection* (i.e., rules that recommend to use an operator in a specific situation), *rejection* (i.e., rules that recommend not to use an operator in a specific situation or avoid a world state), and *preference* (i.e., rules that indicate some operators are preferable in specific situations) rules.

The input for learning how to solve a planning problem as mentioned in Langley (1996) consists of partial given knowledge of a problem-solving domain and traces of search performed by the system in the search space of problems in that domain. The idea that this set of experiences can contain *solution paths* (or in our terminology, plan traces) was suggested in (Sleeman, Langley, and Mitchell 1982). Mitchell, Mahadevan, and Steinbert (1985) suggest *learning apprentices*, which acquire their knowledge by observing a domain expert solving a problem, be used as control rule learning algorithms for the first time.

Explanation-Based Learning (EBL) has been used to induce control rules (Minton 1988). STATIC (Etzioni 1993) uses a graph representation of problem spaces to derive EBL-style control knowledge. Shavlik (1989) introduces BAGGER, a system that uses analytical methods and EBL to learn recursive structures. Kautukam and Kambhampati (1994) discuss the deduction of explanation-based control rules in partial ordered planning. Leckie and Zukerman (1998) uses inductive methods to learn search control rules. Borrajo and Veloso (1997) introduce a lazy learning method that combines deductive and inductive strategies to incrementally learn control knowledge. The knowledge acquired by this method is used to refine and/or override the default behavior of the planner in cases where doing so will improve planner's efficiency or solution quality. Another widely used approach to acquire domain control knowledge in action-based planners is that of learning *Macro-Operators* (i.e., a sequence of operators that frequently appear together in plans and, therefore, can be treated as one superoperator that can achieve the collective effects of those operators) (e.g., see Iba 1989; Mooney 1989; Tadepalli 1991; Tadepalli and Natarajan 1996). In Zelle and Mooney (1993), EBL is used in the context of inductive logic programming to induce clause selection

rules that can significantly speed up logic programs by helping them choose the next branch in the search space to follow.

There has been some recent work on applying various learning algorithms to induce task hierarchies. Garland, Ryall, and Rich (2001) use a technique called *programming by demonstration* to build a system in which a domain expert performs a task by executing actions and then reviews and annotates a log of the actions. This information is then used to learn hierarchical task models. KnoMic (van Lent and Laird 1999) is a learning-by-observation system that extracts knowledge from observations of an expert performing a task and generalizes this knowledge to a hierarchy of rules. These rules are then used by an agent to perform the same task. Langley and Rogers (2004) describe how ICARUS, a cognitive architecture that stores its knowledge of the world in two hierarchical categories of *Concept memory* and *skill memory*, can learn these hierarchies by experiencing problem solving in sample domains. Stone and Veloso (2000) discuss a system that learns a hierarchical controller to play robot soccer. Other systems that deal with hierarchy induction using sample data include MARVIN (Sammut and Banerji 1986), X-Learn (Reddy and Tadepalli 1997), Sequitur-IAM (Pfleger 2002), and STL (Utgoff and Stracuzzi 2002).

Another aspect concerning the integration of planning and learning is *automatic domain knowledge acquisition*. In this framework, the planner does not have the full definition of the planning domain and tries to learn this definition by experimentation. Gil (1992, 1994) discusses a dynamic environment in which the preconditions or effects of operators change over time, and methods to derive these preconditions and effects dynamically. In (Gil 1993), instead of revising existing operators, new operators are acquired by direct analogy with existing operators, decomposition of monolithic operators into meaningful suboperators and experimentation with partially specified operators. These are all done in conventional action-based planning environment rather than HTN as in CaMeL. CaMeL assumes that the complete definition of the planning operators and their add and delete lists are known in advance. It instead focuses on learning HTN methods.

Winner and Veloso (2002, 2003a) use plan examples to learn *plan templates*. These plan templates are pieces of code that compactly represent the domain-specific information learned during the planning process. These plan templates are then merged and generalized to synthesize domain-specific planners that can then be used to solve new problems in that domain. Winner and Veloso (2003b) introduce algorithm DISTILL. DISTILL converts each of its input plans to a domain-specific planner and then merges it with previously learned domain-specific planners. It can also derive one-step loops from example plans that enable it to apply its experience with smaller problems to arbitrarily large ones. This algorithm is used in action-based planning rather than HTN planning as in CaMeL.

PRODIGY (Minton et al. 1989) is an architecture that integrates planning and learning in its several modules (Veloso et al. 1995). SCOPE (Estlin 1998) is a system that learns domain-specific control rules for a partial-ordered planner that improve both planning efficiency and plan quality (Estlin and Mooney 1997) and uses both EBL and Inductive Logic Programming (ILP) techniques. SOAR (Laird, Rosenbloom, and Newell 1986) is a general cognitive architecture for developing systems that exhibit intelligent behavior. These systems are mostly focused on learning control knowledge. In contrast, CaMeL learns domain knowledge.

PASSAT (Myers et al. 2002) is a mixed-initiative, human-centric, HTN-based planning framework that allows users to use its rich library of templates (which are modeled as task networks) to author plans. Final plans are supposed to include *plan sketches* (Myers and Lee, 1997), which are prescriptions of goals and actions to be included in the solution. Myers et al. (2003) discuss mixed-initiative approaches to handle cases where given plan sketches are not consistent. The system guides a human planner through the process of modifying a plan sketch to eliminate detected problems by identifying sketch problems and possible

repairs, while the human acts as the final decision maker. In other words, the main focus here is assisting a human being to author a plan, while CaMeL's focus is on automated planning.

8. CONCLUSION AND FUTURE WORK

In this paper, we have formulated a framework to define and evaluate what an HTN precondition learner is and what it should do. We have extended the definitions of soundness, completeness and convergence in an intuitive way to fit in this framework. We have introduced CaMeL, an algorithm that uses machine learning techniques to complete an incomplete HTN domain description so that it can be used to solve planning problems later. CaMeL is supposed to observe domain experts while they are solving instances of ordered HTN planning problems, and gather and generalize information on how these experts solved these problems, so that it can assist other users in future planning problems. Experiments suggest that CaMeL can quickly (i.e., with a small number of plan traces) learn the methods that are most useful in a planning domain. This suggests that CaMeL may potentially be useful in real-world applications, because our experiments in the NEO domain show that it can generate plans for many problems even before it has fully learned all the methods in a domain, using only the methods it has fully learned to generate plans and ignoring the methods it has not fully learned.

CaMeL is an incremental algorithm. Therefore, even if it has not been given enough training samples to learn all the methods in a domain completely, it should be able to approximate the methods that have not yet been fully learned, and use those approximations in generating plans rather than just ignoring them. Our future work will include developing techniques to do these approximations.

There is more information in a plan trace than CaMeL currently uses. At each decision point, in addition to the list of all applicable methods, CaMeL knows which method was actually chosen to be applied. This latter piece of information can be used to learn rules that prescribe certain applicable methods over other such methods by observing such decisions made by domain experts and recorded in input plan traces. We intend to implement another learning module that derives such rules from plan traces. These rules will be similar to the preference rules in classical planning discussed in the previous Section.

ACKNOWLEDGMENTS

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F306029910013 and F30602-00-2-0505, Naval Research Laboratory S00000322 and S00000849, Army Research Laboratory DAAL0197K0135, and the University of Maryland General Research Board. Opinions expressed in this paper are those of authors and do not necessarily reflect opinions of the funding agencies. We would also like to thank the reviewers for their detailed and helpful comments and suggestions.

REFERENCES

BACCHUS, F. 2000. Aips-00 planning competition. http://www.cs.toronto.edu/aips2000.

BORRAJO, D., and M. VELOSO. 1997. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. Artificial Intelligence Review, Special Issue on Lazy Learning, **11** (1–5):371–405.

CURRIE, K., and A. TATE. 1991. O-plan: The open planning architecture. Artificial Intelligence, **52** (1):49–86.

EDELKAMP, S., and J. HOFFMANN. 2004. International planning competition. http://ls5-www.cs.uni-dortmund.de/∼edelkamp/ipc-4/.

EROL, K., J. HENDLER, and D. S. NAU. 1996. Complexity results for hierarchical task-network planning. Annals of Mathematics and Artificial Intelligence, **18**(1):69–93.

ESTLIN, T. 1998. Using multi-strategy learning to improve planning efficiency and quality. Ph. D. Thesis, Department of Computer Sciences, University of Texas at Austin.

ESTLIN, T. A., and R. J. MOONEY. 1997. Learning to improve both efficiency and quality of planning. *In* Proceedings of the 15th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Nagoya, Japan, pp. 1227–1232.

ETZIONI, O. 1993. A structural theory of explanation-based learning. Artificial Intelligence, **60**(1):93–139.

FOX, M., and D. LONG. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research, Special Issue on the 3rd International Planning Competition, **20**:61–124.

GARLAND, A., K. RYALL, and C. RICH. 2001. Learning hierarchical task models by defining and refining examples. *In* Proceedings of the First International Conference on Knowledge Capture. ACM Press, Victoria, British Columbia, Canada, pp. 44–51.

GHALLAB, M., and H. LARUELLE. 1994. Representation and control in IxTeT, a temporal planner. *In* Proceedings of the Second International Conference on Artificial Intelligence Planning Systems. AAAI Press, Chicago, IL, pp. 61–67.

GHALLAB, M., D. NAU, and P. TRAVERSO. 2004. Automated Planning: Theory and Practice. Morgan Kaufmann, San Francisco, CA.

GIL, Y. 1992. Acquiring domain knowledge for planning by experimentation. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

GIL, Y. 1993. Learning new planning operators by exploration and experimentation. *In* Proceedings of the AAAI Workshop on Learning Action Models. AAAI Press, Washington, DC, pp. 1–4.

GIL, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. *In* Proceedings of the 11th International Conference on Machine Learning. Morgan Kaufmann, New Brunswick, NJ, pp. 87–95.

GORDON, D., and M. DESJARDINS. 1995. Evaluation and selection of biases in machine learning. Machine Learning, **20** (1–2):5–22.

GUPTA, N., and D. NAU. 1992. On the complexity of blocks-world planning. Artificial Intelligence, **56** (2–3):223–254.

HANNEY, K., and M. T. KEANE. 1996. Learning adaptation rules from a case-base. *In* Proceedings of the 3rd European Workshop on Case-Based Reasoning. Springer, Lausanne, Switzerland, pp. 179–192.

HIRSH, H. 1994. Generalizing version spaces. Machine Learning, **17** (1):5–45.

HIRSH, H., N. MISHRA, and L. PITT. 1997. Version spaces without boundary sets. *In* Proceedings of the 14th National Conference on Artificial Intelligence. AAAI Press, Providence, RI, pp. 491–496.

HIRSH, H., N. MISHRA, and L. PITT. 2004. Version spaces and the consistency problem. Artificial Intelligence, **156** (2):115–138.

IBA, G. A. 1989. A heuristic approach to the discovery of macro-operators. Machine Learning, **3** (4):285–317.

KAUTUKAM, S., and S. KAMBHAMPATI. 1994. Learning explanation-based search control rules for partial-order planning. *In* Proceedings of the 12th National Conference on Artificial Intelligence. AAAI Press, Seattle, WA, pp. 582–587.

LAIRD, J. E., P. S. ROSENBLOOM, and A. NEWELL. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. Machine Learning, **1** (1):11–46.

LANGLEY, P. 1996. Elements of Machine Learning. Morgan Kaufmann, San Francisco, CA.

LANGLEY, P., and S. ROGERS. 2004. Cumulative learning of hierarchical skills. *In* Proceedings of the 3rd International Conference on Development and Learning. IEEE Press, La Jolla, CA.

LECKIE, C., and I. ZUKERMAN. 1998. Inductive learning of search control rules for planning. Artificial Intelligence, **101** (1–2):63–98.

LENZ, M., B. BARTSCH-SPORL, H. D. BURKHARD, and S. WESS. (Eds.). 1998. Case-Based Reasoning Technology: From Foundations to Applications. Springer, Berlin, Germany.

LONG, D., and M. FOX. 2002. International planning competition. http://www.cis.strath.ac.uk/∼derek/competition.html.

MCDERMOTT, D. 1998. PDDL, the planning domain definition language. Technical Report 1165, Yale Center for Computational Vision and Control, New Haven, CT.

MINTON, S. N. 1988. Learning effective search control knowledge: An explanation-based approach. Technical Report TR CMU-CS-88-133, School of Computer Science, Carnegie Mellon University.

MINTON, S. N. 1990. Quantitative results concerning the utility of explanation based learning. Artificial Intelligence, **42** (2–3):363–391.

MINTON, S. N., C. A. KNOBLOCK, D. R. KUOKKA, Y. GIL, R. L. JOSEPH, and J. G. CARBONELL. 1989. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University.

MITCHELL, T. M. 1977. Version spaces: A candidate elimination approach to rule learning. *In* Proceedings of the 5th International Joint Conference on Artificial Intelligence. William Kaufmann, Cambridge, MA, pp. 305–310.

MITCHELL, T. M. 1980. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University.

MITCHELL, T. M., S. MAHADEVAN, and L. STEINBERG. 1985. LEAP: A learning apprentice for VLSI design. *In* Proceedings of the 9th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Los Angeles, CA, pp. 573–580.

MOONEY, R. J. 1989. The effect of rule use on the utility of explanation-based learning. *In* Proceedings of the 11th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Detroit, MI, pp. 725–730.

MUÑOZ-AVILA, H., D. MCFARLANE, D. W. AHA, J. BALLAS, L. A. BRESLOW, and D. S. NAU. 1999. Using guidelines to constrain interactive case-based HTN planning. *In* Proceedings of the 3rd International Conference on Case-Based Reasoning and Development. Springer Press, Munich, Germany, pp. 288–302.

MUSCETTOLA, N., P. P. NAYAK, B. PELL, and B. C. WILLIAMS. 1998. Remote agent: To boldly go where no AI system has gone before. Artificial Intelligence, **103** (1–2):5–47.

MYERS, K., and T. J. LEE. 1997. Abductive completion of plan sketches. *In* Proceedings of the 14th National Conference on Artificial Intelligence. AAAI Press, Providence, RI, pp. 687–693.

MYERS, K. L., P. A. JARVIS, W. M. TYSON, and M. J. WOLVERTON. 2003. A mixed-initiative framework for robust plan sketching. *In* Proceedings of the 13th International Conference on Automated Planning and Scheduling. AAAI Press, Trento, Italy, pp. 256–266.

MYERS, K. L., W. M. TYSON, M. J. WOLVERTON, P. A. JARVIS, T. J. LEE, and M. DESJARDINS. 2002. PASSAT: A user-centric planning framework. *In* Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space. Houston, TX.

NAU, D. S., T.-C. AU, O. ILGHAMI, U. KUTER, H. MUÑOZ-AVILA, J. W. MURDOCK, D. WU, and F. YAMAN. 2004. Applications of shop and shop2. Technical Report CS-TR-4604, Department of Computer Science, University of Maryland.

NAU, D. S., Y. CAO, A. LOTEM, and H. MUÑOZ-AVILA. 1999. SHOP: Simple hierarchical ordered planner. *In* Proceedings of the 16th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Stockholm, Sweden, pp. 968–973.

NAU, D. S., S. J. J. SMITH, and K. EROL. 1998. Control strategies in HTN planning: Theory versus practice. *In* The 15th National Conference on Artificial Intelligence. AAAI Press, Madison, Wisconsin, pp. 1127–1133.

PFLEGER, K. 2002. On-line learning of predictive compositional hierarchies. Ph. D. Thesis, Computer Sciences Department, Stanford University, Stanford, CA.

REDDY, C., and P. TADEPALLI. 1997. Learning goal-decomposition rules using exercises. *In* Proceedings of the 14th International Conference on Machine Learning. Morgan Kaufmann, Nashville, TN, pp. 278–286.

SACERDOTI, E. 1975. The nonlinear nature of plans. *In* The Proceedings of the 4th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Tiblisi, USSR, pp. 206–214.

SAMMUT, C., and R. B. BANERJI. 1986. Leaning concepts by asking questions. *In* Machine Learning: An Artificial Intelligence Approach, Volume 2. *Edited by* R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. Morgan Kaufmann, Los Altos, CA.

SEBAG, M. 1995. 2nd order understandability of disjunctive version spaces. *In* Workshop on Machine Learning and Comprehensibility, 14th International Joint Conference on Artificial Intelligence. Montreal, Quebec, Canada, pp. 356–365.

SHAVLIK, J. W. 1989. Acquiring recursive concepts with explanation based learning. *In* Proceedings of the 11th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Detroit, MI, pp. 688–693.

SLEEMAN, D., P. LANGLEY, and T. M. MITCHELL. 1982. Learning from solution paths: An approach to the credit assignment problem. AI Magazine, **3**(2):48–52.

SMITH, S. J. J., D. S. NAU, and T. THROOP. 1998. Computer bridge: A big win for AI planning. AI Magazine, **19**(2):93–105.

STONE, P., and M. M. VELOSO. 2000. Layered learning. *In* Proceedings of the 11th European Conference on Machine Learning. Springer, Barcelona, Spain, pp. 369–381.

TADEPALLI, P. 1991. A formalization of explanation-based macro-operator learning. *In* Proceedings of the 12th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Sydney, Australia, pp. 616–622.

TADEPALLI, P., and B. K. NATARAJAN. 1996. A formal framework for speedup learning from problems and solutions. Journal of Artificial Intelligence Research, **4**:445–475.

TATE, A. 1977. Generating project networks. *In* Proceedings of the 5th International Joint Conference on Artificial Intelligence. MIT Press, Cambridge, MA, pp. 888–893.

UTGOFF, P., and D. STRACUZZI. 2002. Many-layered learning. *In* Proceedings of the 2nd International Conference on Development and Learning. IEEE Press, Cambridge, MA, pp. 141–146.

VAN LENT, M., and J. LAIRD. 1999. Learning hierarchical performance knowledge by observation. *In* Proceedings of the 16th International Conference on Machine Learning. Morgan Kaufmann, Bled, Slovenia, pp. 229–238.

VELOSO, M., and J. G. CARBONELL. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. Machine Learning, **10**(3):249–278.

VELOSO, M., J. G. CARBONELL, A. PÉREZ, D. BORRAJO, E. FINK, and J. BLYTHE. 1995. Integrating planning and learning: The PRODIGY architecture. Journal of Experimental and Theoretical Artificial Intelligence, **7**(1):81–120.

VELOSO, M., H. MUÑOZ-AVILA, and R. BERGMANN. 1996. Case-based planning: Selected methods and systems. AI Communications, **9**(3):128–137.

WILKINS, D. E. 1990. Can AI planners solve practical problems? Computational Intelligence, **6**(4):232–246.

WINNER, E., and M. VELOSO. 2002. Automatically acquiring planning templates from example plans. *In* Proceedings of the Workshop on Exploring Real-World Planning. Toulouse, France, pp. 69–74.

WINNER, E., and M. VELOSO. 2003a. Acquiring domain-specific planners by example. Technical Report CMU-CS-03-101, Carnegie Mellon Computer Science Department, Pittsburgh, PA.

WINNER, E., and M. VELOSO. 2003b. DISTILL: Learning domain-specific planners by example. *In* Proceedings of the 20th International Conference on Machine Learning. AAAI Press, Washington, DC, pp. 800–807.

ZELLE, J. M., and R. J. MOONEY. 1993. Combining FOIL and EBG to speed up logic programs. *In* Proceedings of the 13th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Chambery, France, pp. 1106–1111.