Technical report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, March, 1994.

Semantics for Hierarchical Task-Network Planning*

Kutluhan Erol kutluhan@cs.umd.edu
James Hendler hendler@cs.umd.edu
Dana S. Nau nau@cs.umd.edu

Institute for Advanced Computer Studies,
Institute for Systems Research,
Computer Science Department,
University of Maryland, College Park, MD 20742

Abstract

One big obstacle to understanding the nature of hierarchical task network (HTN) planning has been the lack of a clear theoretical framework. In particular, no one has yet presented a clear and concise HTN algorithm that is sound and complete. In this paper, we present a formal syntax and semantics for HTN planning. Based on this syntax and semantics, we are able to define an algorithm for HTN planning and prove it sound and complete. We also develop several definitions of expressivity for planning languages and prove that HTN planning is strictly more expressive than STRIPS-style planning according to those definitions.

1 Introduction

In AI planning research, planning practice (as embodied in implemented planning systems) tends to run far ahead of the theories that explain the behavior of those planning systems. For

*This work was supported in part by NSF Grants IRI-9306580 and NSF EEC 94-02384, AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), and ONR grant N00014-91-J-1451. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

example, STRIPS-style planning systems[1] were developed more than twenty years ago (Fikes *et al.*, 1971), and most of the practical work on AI planning systems for the last fifteen years has been based on hierarchical task-network (HTN) decomposition (Sacerdoti, 1977; Tate, 1990; Wilkins, 1988a; Wilkins, 1988b). In contrast, although the past few years have seen much analysis of planning using STRIPS-style operators, (Chapman, 1987; Mcallester *et al.*, 1991; Erol *et al.*, 1992b; Erol *et al.*, 1992a), there has been very little analytical work on HTN planners.

One big obstacle to such work has been the lack of a clear theoretical framework for HTN planning. Two recent papers (Yang, 1990; Kambhampati *et al.*, 1992) have provided important first steps towards formalizing HTN planning, but these focused on the syntax, rather than the semantics. As a result, no one has presented a clear and concise HTN algorithm that is sound and complete. In this paper, we present a syntax *and* semantics for HTN planning, which enables further analytical work. In particular, our formalism allows us to evaluate the expressive power of HTN planning and develop correct HTN planning algorithms.

The paper is organized as follows. Section 2 contains an informal overview of HTN planning. In Section 3, we present our formalism, and in Section 4 we describe how various features of HTN planning fit into our formalism. Section 5 contains our provably sound and complete planning algorithm **UMCP**. In Section 6, we develop several definitions of expressivity for planning languages, and prove that HTN planning is more powerful than STRIPS-style planning according to those definitions.

2 An Overview of HTN planning

This section contains an informal description intended to give an intuitive feel for HTN planning. A precise formal description is presented in Section 3.

One of the motivations for HTN planning was to close the gap between AI planning techniques such as STRIPS-style planning, and operations-research techniques for project management and scheduling (Tate, 1990). Thus, there are some similarities between HTN planning and STRIPS-style planning, but also some significant differences.

HTN planning uses actions and states of the world that are similar to those used in STRIPS-style planning.[2] Each state of the world is represented by the set of atoms true in

[1]We will refer to planning systems that use STRIPS operators (with no decompositions) as STRIPS-style planners, ignoring algorithmic differences among them that are not relevant to the current work.

[2]We use the term "STRIPS-style" planning to refer to any planner (either total- or partial-order) in which the planning operators are STRIPS-style operators (i.e., operators consisting of three lists of atoms: a precondition list, an add list, and a delete list). These atoms are normally assumed to contain no function

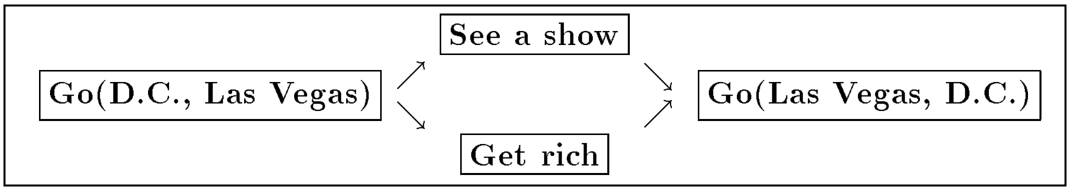

Figure 1: A task network

that state. Actions, which in HTN planning are usually called *primitive tasks*, correspond to state transitions; i.e., each action is a partial mapping from the set of states to set of states.

The primary difference between HTN planners and STRIPS-style planners is in what they plan for, and how they plan for it. In STRIPS-style planning, the objective is to find a sequence of actions that will bring the world to a state that satisfies certain conditions or "attainment goals." Planning proceeds by finding operators that have the desired effects, and by making the preconditions of those operators into subgoals. In contrast, HTN planners search for plans that accomplish *task networks*, which can include things other than just attainment goals; and they plan via task decomposition and conflict resolution, which we shall explain shortly.

A task network is a collection of tasks that need to be carried out, together with constraints on the order in which tasks can be performed, the way variables are instantiated, and what literals must be true before or after each task is performed. For example, Figure 1 contains a task network for a trip to Las Vegas. Unlike STRIPS-style planning, the constraints may or may not contain conditions on what must be true in the final state.

A task network that contains only primitive tasks is called a *primitive task network*. Such a network might occur, for example, in a scheduling problem. In this case, an HTN planner would try to find a schedule (task ordering and variable bindings) that satisfies all the constraints.

In the more general case, a task network can contain *non-primitive tasks*, which the planner needs to figure out how to accomplish. Non-primitive tasks cannot be executed directly, because they represent activities that may involve performing several other tasks. For example the task of traveling to New York can be accomplished in several ways, such as flying, driving or taking the train. Flying would involve tasks such as making reservations, going to the airport, buying ticket, boarding the plane; and flying would only work if certain conditions were satisfied, such as availability of tickets, being at the airport on time, having enough money for the ticket, and so forth.

symbols.

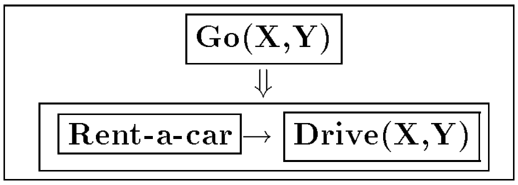

Figure 2: A (simplified) method for going from X to Y.

1. Input a planning problem **P**.
2. If **P** contains only primitive tasks, then
   resolve the conflicts in **P** and return the result.
   If the conflicts cannot be resolved, return failure.
3. Choose a non-primitive task t in **P**.
4. Choose an expansion for t.
5. Replace t with the expansion.
6. Use critics to find the interactions among the tasks in **P**,
   and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

Figure 3: The basic HTN Planning Procedure.

Ways of accomplishing non-primitive tasks are represented using constructs called *methods*. A method is a syntactic construct of the form (α, d) where α is a non-primitive task, and d is a task network. It states that one way to accomplish the task α is to achieve all the tasks in the task network d without violating the constraints in d. Figure 2 presents a (simplified) method for accomplishing **Go(X,Y)**.

A number of different systems that use heuristic algorithms have been devised for HTN planning (Tate, 1990; Vere, 1983; Wilkins, 1988a), and several recent papers have tried to provide formal descriptions of these algorithms (Yang, 1990; Kambhampati *et al.*, 1992). Figure 3 presents the essence of these algorithms. As shown in this figure, HTN planning works by expanding tasks and resolving conflicts iteratively, until a conflict-free plan can be found that consists only of primitive tasks.

Expanding or *reducing* each non-primitive task (steps 3–5) is done by finding a method

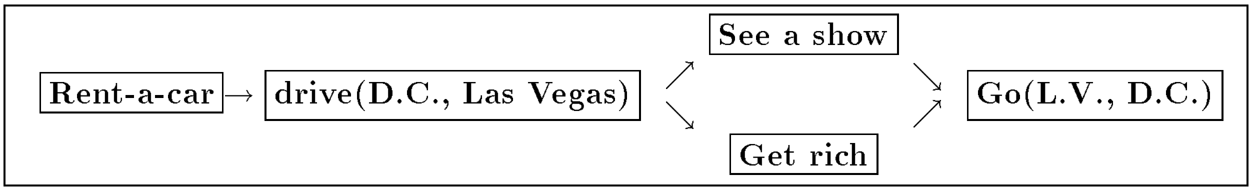

Figure 4: A decomposition of the the task network in Fig. 1

capable of accomplishing the non-primitive task, and replacing the non-primitive task with the task network produced by the method. For example, the task **Go(D.C., Las Vegas)** in the task network of Figure 1 can be expanded using the method in Figure 2, producing the task network in Figure 4.

The task network produced in Step 5 may contain conflicts caused by the interactions among tasks. For example, in Figure 4, if we use up all our money in order to rent the car, we may not be able to see a show. The job of finding and resolving such interactions is performed by critics. Historically speaking, critics were introduced into NOAH (Sacerdoti, 1977) to identify and deal with several kinds of interactions (not just deleted preconditions) among the different networks used to reduce each non-primitive operator. This is reflected in Steps 6 and 7 of Figure 3: after each reduction, a set of critics is checked so as to recognize and resolve interactions between this and any other reductions. Thus, critics provide a general mechanism for detecting interactions early, so as to reduce the amount of backtracking. For a more detailed discussion of the many different ways critic functions have been used, see (Tate *et al.*, 1990).

3 HTN Formalism

Although the basic idea of HTN planning has been around since 1974, the lack of a clear theoretical foundation has made it difficult to explore its properties. In particular, although it is easy to state the algorithm shown in Figure 3, proving it sound and complete requires considerable formal development. Below, we present a syntax and semantics for HTN planning.

3.1 Syntax for HTN Planning

Our language $\mathcal{L}$ for HTN planning is a first-order language with some extensions, and it is fairly similar to the syntax of NONLIN (Tate, 1990). The vocabulary of $\mathcal{L}$ is a tuple

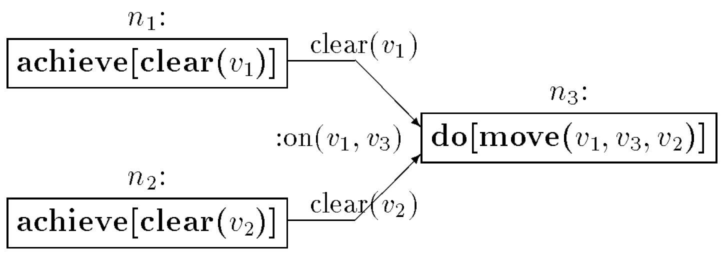

Figure 5: Graphical representation of a task network.

$\langle V, C, P, F, T, N\rangle$, where V is an infinite set of variable symbols, C is a finite set of constant symbols, P is a finite set of predicate symbols, F is a finite set of *primitive*-task symbols (denoting actions), T is a finite set of *compound*-task symbols, and N is an infinite set of symbols used for labeling tasks. All these sets of symbols are mutually disjoint.

A *state* is a list of ground atoms. The atoms appearing in that list are said to be true in that state and those that do not appear are false in that state.

A *primitive task* is a syntactic construct of the form $do[f(x_1, \ldots, x_k)]$, where $f \in F$ and $x_1, \ldots, x_k$ are terms. A *goal task* is a syntactic construct of the form $achieve[l]$, where l is a literal. A *compound task* is a syntactic construct of the form $perform[t(x_1, \ldots, x_k)]$, where $t \in T$ and $x_1, \ldots, x_k$ are terms. We sometimes refer to goal tasks and compound tasks as non-primitive tasks.

A *plan* is a sequence σ of ground primitive tasks.

A *task network* is a syntactic construct of the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$, where

- each α_i is a task;
- $n_i \in N$ is a label for α_i (to distinguish it from any other occurrences of α_i in the network);
- ϕ is a boolean formula constructed from variable binding constraints such as $(v = v')$ and $(v = c)$, ordering constraints such as $(n \prec n')$, and state constraints such as (n, l), (l, n), and (n, l, n'), where $v, v' \in V$, l is a literal, $c \in C$, and $n, n' \in N$.[3] Intuitively (this will be formalized in Section 3.2, $(n \prec n')$ means that the task labeled with n must precede the one labeled with n'; (n, l), (l, n) and (n, l, n') mean that l must be true in the state immediately after n, immediately before n, and in all states between n and n', respectively. Both negation and disjunction are allowed in the constraint formula.

[3] We also allow n, n' to be of the form $first[n_i, n_j, \ldots]$ or $last[n_i, n_j, \ldots]$ so that we can refer to the task that starts first and to the task that ends last among a set of tasks, respectively.

$$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])(n_3 : do[move(v_1, v_3, v_2)])$$
$$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3) \wedge (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3)$$
$$\wedge \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

Figure 6: Formal representation of the task network of Fig. 5.

A task network containing only primitive tasks is called a *primitive task network*.

As an example, Fig. 6 gives a formal representation of the task network shown in Figure 5. In this blocks-world task network there are three tasks: clearing v_1, clearing v_2, and moving v_1 to v_2. The task network also includes the constraints that moving v_1 must be done last, that v_1 and v_2 must remain clear until we move v_1, that v_1, v_2, v_3 are different blocks, and that $on(v_1, v_3)$ be true immediately before v_1 is moved. Note that $on(v_1, v_3)$ appears as a constraint, not as a goal task. The purpose of the constraint $(on(v_1, v_3), n_3)$ is to ensure that v_3 is bound to the block under v_1 immediately before the move. Representing $on(v_1, v_3)$ as a goal task would mean moving v_1 onto some block v_3 before we move it onto v_2, which is not what is intended.

An *operator* is a syntactic construct of the form

$$[operator\ \ f(v_1, \ldots, v_k)(\text{pre:}l_1, \ldots, l_m)(\text{post:}l'_1, \ldots, l'_n)],$$

where f is a primitive task symbol, and $l_1, \ldots, l_m$ are literals describing when f is executable, $l'_1, \ldots, l'_n$ are literals describing the effects of f, and $v_1, \ldots, v_k$ are the variable symbols appearing in the literals.

A *method* is a construct of the form (α, d) where α is a non-primitive task, and d is a task network. As we will define formally in Section 3.2, this construct means that one way of accomplishing the task α is to accomplish the task network d, i.e. to accomplish all the subtasks in the task network without violating the constraint formula of the task network. For example, a blocks-world method for achieving $on(v_1, v_2)$ would look like $(achieve(on(v_1, v_2)), d)$, where d is the task network in Fig. 6. To accomplish a goal task $(achieve[l])$, l needs to be true in the end, and this is an implicit constraint in all methods for goal tasks. If a goal is already true, then an empty plan can be used to achieve it. Thus, for each goal task, we (implicitly) have a method $(achieve[l], [(n : do[f]), (l, n)])$ which contains only one dummy primitive task f with no effects, and the constraint that the goal l is true immediately before $do[f]$.

Each primitive task has exactly one operator for it, where as a non-primitive task can have an arbitrary number of methods.

A *planning domain* $\mathcal{D}$ is a pair $\langle Op, Me \rangle$, where Op is a list of operators, and Me is a list of methods. A *planning problem instance* **P** is a triple $\langle d, I, \mathcal{D}\rangle$, where $\mathcal{D}$ is a planning domain, I is the initial state, and d is the task network we need to plan for. $solves(\sigma, d, I)$ is a syntactic construct which we will use to mean that σ is a plan for the task network d at state I.

3.2 Model-Theoretic Semantics

Before we can develop a sound and complete planning algorithm for HTN planning, we need a semantics that provides meaning to the syntactic constructs of the HTN language, which in turn would define the set of plans for a planning problem.

Semantic Structure

A semantic structure for HTN planning is a triple $M = \langle \mathcal{S}_M, \mathcal{F}_M, \mathcal{T}_M\rangle$. Whenever it is clear from context which model we are referring to, we will say $M = \langle \mathcal{S}, \mathcal{F}, \mathcal{T}\rangle$, omitting the subscript M. $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{T}$ are described below.

$\mathcal{S} = 2^{\{\text{all ground atoms}\}}$ is the set of states. Each state in $\mathcal{S}$ is a set, consisting of the atoms true in that state. Any atom not appearing in a state is considered to be false in that state. Thus, a state corresponds to a "snapshot" instance of the world.

$\mathcal{F} : F\text{x}C^*\text{x}\mathcal{S} \rightarrow \mathcal{S}$ is a partial function for interpreting the actions. Given a primitive task symbol from F, with ground parameters from C, and an input state, $\mathcal{F}$ tells us which state we would end up with, if we were to execute the action. For a given action, $\mathcal{F}$ might be undefined for some input states, namely those for which the action is not executable.

$\mathcal{T} : \{\text{ground non-primitive tasks}\} \rightarrow 2^{\{\text{ground primitive task networks}\}}$ is a function that maps each non-primitive task α to a (not necessarily finite) set of *ground primitive* task networks $\mathcal{T}(\alpha)$. Each primitive task network d in $\mathcal{T}(\alpha)$ contains a set of actions that would achieve α under certain conditions (as specified in the constraint formula of d). There are two restrictions on the way $\mathcal{T}()$ interprets a goal task $achieve[l]$. First, l must be true at the end of any task network in $\mathcal{T}(achieve[l])$. Second, since an empty plan can be used to accomplish a goal task if the goal literal is already true, $\mathcal{T}(achieve[l])$ must contain a task network consisting of a single dummy task with the constraint that l is true.

Task networks can be interpreted similarly to non-primitive tasks, and we extend the domain of $\mathcal{T}$ to cover task networks as follows:

- $\mathcal{T}(\alpha) = \{[(n : \alpha), TRUE]\}$, if α is a ground primitive task. Thus, in order to accomplish α, it suffices to execute it (provided that it *is* executable).

- $\mathcal{T}(d) = \{d\}$, if d is a ground primitive task network.

- To accomplish a non-primitive task network d, we need to accomplish each task in d without violating the constraint formula. Thus we define $\mathcal{T}(d)$ as follows. Let $d = [(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \phi]$ be a ground task network possibly containing non-primitive tasks. Then

  $$\mathcal{T}(d) = \{compose(d_1, \ldots, d_m, \phi) \mid d_i \in \mathcal{T}(\alpha_i),\ i = 1 \ldots m\},$$

  where *compose* is defined as follows. Suppose

  $$d_i = [(n_{i1} : \alpha_{i1}) \ldots (n_{ik_i} : \alpha_{ik_i}), \phi_i]$$

  for each i. Then[4]

  $$compose(d_1, \ldots, d_m, \phi) = [(n_{11} : \alpha_{11}) \ldots (n_{mk_m} : \alpha_{mk_m}), \phi_1 \wedge \ldots \phi_m \wedge \phi'],$$

  where ϕ' is obtained from ϕ by making the following replacements:

  - replace $(n_i < n_j)$ with $(last[n_{i1}, \ldots, n_{ik_i}] < first[n_{j1}, \ldots, n_{jk_j}])$, since all tasks in the decomposition of n_i must precede all tasks in the decomposition of n_j;
  - replace (l, n_i) with $(l, first[n_{i1}, \ldots, n_{ik_i}])$, since l needs to be true immediately before the first task in the decomposition of n_i;
  - replace (n_i, l) with $(last[n_{i1}, \ldots, n_{ik_i}], l)$;
  - replace (n_i, l, n_j) with $(last[n_{i1}, \ldots, n_{ik_i}], l, first[n_{j1}, \ldots, n_{jk_j}])$;
  - everywhere that n_i appears in ϕ in a $first[]$ or a $last[]$ expression, replace it with $n_{i1}, \ldots, n_{ik_i}$.

- $\mathcal{T}(d) = \bigcup_{d' \text{ is a ground instance of } d} \mathcal{T}(d')$, if d is a task network containing variables.

Satisfaction

In this section we describe how syntactic expressions such as operators and methods take truth values in a given model. We will use the phrases "...is true in model M" and "...is satisfied by model M" interchangeably.

A model M satisfies an operator if M interprets the primitive task associated with the operator so that the primitive task is executable under the conditions specified in the preconditions of the operator, and has the effects specified in the postconditions of the operator.

[4]Actually, the formula is slightly more complicated than what is shown, because the variables and node labels in each d_i must be renamed so that no common variable or node label occurs.

More formally, we will say that an operator $[f(v_1, \ldots, v_k)(\text{pre:}l_1, \ldots, l_m)(\text{post:}l'_1, \ldots, l'_n)]$ is *satisfied* by a model M iff for any ground substitution θ and any state s, $\mathcal{F}_m$ has the following properties, where E_n, E_p are the sets of negative and positive literals in $l'_1, \ldots, l'_n$, respectively:

- if $l_1\theta, \ldots, l_m\theta$ are true in s, then $\mathcal{F}_M(f, v_1\theta, \ldots, v_k\theta, s) = (s - E_n\theta) \cup E_p\theta$;
- otherwise, $\mathcal{F}_M(f, v_1\theta, \ldots, v_k\theta, s)$ is undefined.

Next, we want to define the conditions under which a model M satisfies $solves(\sigma, d, s)$, i.e., the conditions under which σ is a plan that accomplishes the task network d starting at state s, in M. First we consider the case where d is primitive.

Let M be a model, d $= [(n_1 : \alpha_1) \cdots (n_{m'} : \alpha_{m'}), \phi]$ be a ground primitive task network, s_0 be a state, and $\sigma = (f_1(c_{11}, \ldots, c_{1k_1}), \ldots, f_m(c_{m1}, \ldots, c_{mk_m}))$ be a plan executable at s_0. Thus, $s_i = \mathcal{F}_M(f_i, c_{i1}, \ldots, c_{ik_i}, s_{i-1})$ for $i = 1 \ldots m$, which are the intermediate states, are all well-defined. We define *a matching* π from d to σ to be a one-to-one function from $\{1, \ldots, m'\}$ to $\{1, \ldots, m\}$ such that whenever $\pi(i) = j$, $\alpha_i = do[f_j(c_{j1}, \ldots, c_{jk_j})]$. Thus a matching provides a total ordering on the tasks. M satisfies $solves(\sigma, d, s)$ if $m = m'$, and there exists a matching π that makes the constraint formula ϕ true. The constraint formula is evaluated as follows:

- $(c_i = c_j)$ is true, if c_i, c_j are the same constant symbols;
- $first[n_i, n_j, \ldots]$ evaluates to $min\{\pi(i), \pi(j), \ldots\}$;
- $last[n_i, n_j, \ldots]$ evaluates to $max\{\pi(i), \pi(j), \ldots\}$;
- $(n_i \prec n_j)$ is true if $\pi(i) < \pi(j)$;
- (l, n_i) is true if l holds in $s_{\pi(i)-1}$;
- (n_i, l) is true if l holds in $s_{\pi(i)}$;
- (n_i, l, n_j) is true if l holds for all s_e, $\pi(i) \leq e < \pi(j)$;
- logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic.

Let d be a task network, possibly containing non-primitive tasks. A model M satisfies $solves(\sigma, d, s)$ if for some $d' \in \mathcal{T}_M(d)$, M satisfies $solves(\sigma, d', s)$.

For a method (α, d) to be satisfied by a given model, not only must any plan for d also be a plan for α, but in addition, any plan for a task network tn containing d must be a plan for the task network obtained from tn by replacing d with α. Thus, a method (α, d) is satisfied by a model M, iff $\mathcal{T}_M(\alpha)$ *covers* $\mathcal{T}_M(d)$, where "covers" is defined as follows:

Given a model M, two sets of ground primitive task networks TN and TN', TN is said to cover TN', iff for any state s, any plan σ executable at s, and any $d' \in TN'$, the following property holds:

> Whenever there exists a matching π between d' and σ such that σ at s satisfies the constraint formula of d', then there exists a $d \in TN$ such that for some matching π' with the same range as π, σ at s makes the constraint formula of d true.

A model M satisfies a planning domain $\mathcal{D} = \langle Op, Me \rangle$, if M satisfies all operators in Op, and all methods in Me.

3.3 Proof Theory

A plan σ solves a planning problem $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$ if any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$. However, given a planning problem, how do we find plans that solve it?

Let d be a primitive task network (one containing only primitive tasks), and let I be the initial state. A plan σ is a *completion* of d at I, denoted by $\sigma \in comp(d, I, \mathcal{D})$, if σ is executable (i.e. the preconditions of each action in σ are satisfied) and σ corresponds to a total ordering of the primitive tasks in a ground instance of d that satisfies the constraint formula of d. For non-primitive task networks d, $comp(d, I, \mathcal{D})$ is defined to be $\emptyset$.

Let d be a non-primitive task network that contains a (non-primitive) node $(n : \alpha)$. Let $m = (\alpha', d')$ be a method, and θ be the most general unifier of α and α'. We define $reduce(d, n, m)$ to be the task network obtained from $d\theta$ by replacing $(n : \alpha)\theta$ with the task nodes of $d'\theta$, modifying the constraint formula ϕ of $d'\theta$ into ϕ' (as we did for *compose*), and incorporating $d'\theta$'s constraint formula. We denote the set of reductions of d by $red(d, I, \mathcal{D})$. Reductions formalize the notion of *task decomposition*.

A plan σ solves a primitive task network d at initial state I iff $\sigma \in comp(d, I, \mathcal{D})$; a plan σ solves a non-primitive task network d at initial state I iff σ solves some reduction $d' \in red(d, I, \mathcal{D})$ at initial state I.

Now, we can define the set of plans $sol(d, I, \mathcal{D})$ that solves a planning problem instance $\mathbf{P} =< d, I, \mathcal{D} >$:

$$
\begin{array}{lcl}
sol_1(d, I, \mathcal{D}) & = & comp(d, I, \mathcal{D}) \\
sol_{n+1}(d, I, \mathcal{D}) & = & sol_n(d, I, \mathcal{D}) \cup \bigcup_{d' \in red(d, I, \mathcal{D})} sol_n(d', I, \mathcal{D}) \\
sol(d, I, \mathcal{D}) & = & \cup_{n < \omega} sol_n(d, I, \mathcal{D})
\end{array}
$$

Intuitively, $sol_n(d, I, \mathcal{D})$ is the set of plans that can be derived in n steps, and $sol(d, I, \mathcal{D})$ is the set of plans that can be derived in any finite number of steps.

In Section 3.2, we presented a model-theoretic semantics for HTN planning, and in this section we have presented an operational, fixed-point semantics that provides a procedural characterization of the set of solutions to planning problems. The next step is to show that the model-theoretic semantics and operational semantics are equivalent, so that we can use the model-theoretic semantics to get a precise understanding of HTN planning, and use the operational semantics to build sound and complete planning systems. The following theorem states that $sol(d, I, \mathcal{D})$ is indeed the set of plans that solves $\langle d, I, \mathcal{D}\rangle$.

Theorem 1 (Equivalence Theorem) *Given a task network d, an initial state I, and a plan σ, σ is in $sol(d, I, \mathcal{D})$ iff any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$.*

This theorem follows from the fact that $sol(d, I, \mathcal{D})$ is constructed such that it always contains only the plans for a task network d with respect to the minimum model. We prove the theorem by constructing a model M such that for any non-primitive task α, $\mathcal{T}_M(\alpha)$ contains the primitive task networks that can be obtained by a finite number of reduction steps from α. Then we prove M to be the minimum model satisfying $\mathcal{D}$. The details of the proof are in the appendix.

Since the set of plans that can be derived using **R1** and **R2** is exactly $sol(d, I, \mathcal{D})$, the corollary immediately follows from the equivalence theorem.

4 Features of HTN Planning

Using the syntax and semantics as defined above, we can now define and/or explain a number of items that are often discussed in the literature:

Tasks and Task-decomposition

There appears to be some general confusion about the nature and role of tasks in HTN planning. This seems largely due to the fact that HTN planning emerged, without a formal description, in implemented planning systems (Sacerdoti, 1977; Tate, 1990). Many ideas introduced in HTN planning (such as nonlinearity, partial order planning, etc.) were formalized only as they were adapted to STRIPS-style planning, and only within that context (Chapman, 1987; Mcallester *et al.*, 1991; Minton *et al.*, 1991; Barett *et al.*, 1992; Collins *et al.*, 1992; Kambhampati, 1992). Those ideas not adapted to STRIPS-style planning (such as compound tasks and task decomposition) have even been dismissed as mere efficiency hacks. In fact, one view of HTN planning totally discards compound tasks, and views methods for goal tasks as heuristic information on how to go about achieving the goals (i.e., which operator to use, in which order to achieve the preconditions of that operator). Although this is a perfectly coherent view, we find it restrictive, and we believe there is more to HTN planning, as we try to demonstrate in our formalism and in the section on expressive power.

We view tasks as activities we need to plan i.e. things that need to be accomplished. Each method tells us one way of achieving a task, and it also tells us under which conditions that way is going to succeed (as expressed in the constraint formula) Finally, the task decomposition refers to choosing a method for the task, and using it to achieve the task. For example, possible methods for the task of traveling to a city might be flying (under the condition that airports are not closed due to bad weather), taking the train (under the condition that there is an available ticket), or renting a car (under the condition that I have a driver's license). Tasks and task networks provide a more natural and flexible way of representing planning problems than STRIPS-style attainment goals, as Lansky (Lansky, 1988) argues for action-based planning in general.

Our formalism is mostly shaped after NONLIN (Tate, 1990) and the works of Yang and Kambhampati (Yang, 1990; Kambhampati *et al.*, 1992) on hierarchical planning. However, our terminology for referring to compound tasks is slightly different from theirs, which instead uses the term "high level actions" (Sacerdoti, 1977; Yang, 1990). Although this term has some intuitive appeal, we prefer not to use it, in order to avoid any possible confusion with STRIPS-style actions. STRIPS-style actions are atomic, and they always have the same effect on the world; non-primitive tasks can be decomposed into a number of primitive tasks, and the effect of accomplishing a non-primitive task depends not only on the methods chosen for doing decompositions, but also on the interleavings with other tasks. For example, consider the task of "round-trip to New York". The amount of money I have got after the trip depends on whether I flew or took a train, and also on my activities in New York (night clubs, etc). Unlike STRIPS-style actions, tasks are not "executed" for their effects in the world, but they are ends by themselves. In this aspect, tasks are more similar to STRIPS-style goals than STRIPS-style actions.

Here are some more examples to further clarify the distinctions among different types of tasks and STRIPS-style goals. Building a house requires many other tasks to be performed (laying the foundation, building the walls, etc.), thus it is a compound task. It is different from the goal task of "having a house," since buying a house would achieve this goal task, but not the compound task of building a house (the agent must build it himself). As another example, the compound task of making a round trip to New York cannot easily be expressed as a single goal task, because the initial and final states would be the same. Goal tasks are very similar to STRIPS-style goals. However, in STRIPS-style planning, *any* sequence of actions that make the goal expression true is a valid plan, where as in HTN planning, only those plans that can be derived via decompositions are considered as valid. This allows the user to rule out certain undesirable sequences of actions that nonetheless make the goal expression true. For example, consider the goal task of "being in New York", and suppose the planner is investigating the possibility of driving to accomplish this goal, and suppose that the agent does not have a driver's license. Even though learning how to drive and getting a driver's license might remedy the situation, the user can consider this solution unacceptable, and while writing down the methods for **be-in(New York)**, add the constraint that the method of driving succeeds only when the agent already has a driver's license.

High-Level Effects

Typically, HTN planners allow one to attach "high-level" effects to subtasks in methods, similar to the way we attach effects to primitive tasks using operators. Some HTN-planners such as NONLIN assume that the high-level effects will be true immediately after the corresponding subtasks, even if they are not asserted by any primitive tasks. This is problematic: one can obtain the same sequence of primitive tasks with different tasks and methods, and given high-level effects, the final state might depend on what particular task(s) the sequence was intended for. Yang (Yang, 1990) addresses this problem by attaching high-level effects to tasks directly using operators. In addition, he requires that for each high-level effect l associated with a task α, every decomposition of α must contain a subtask with the effect l, which is not clobbered by any other subtask in the same decomposition. However, this solution does not preclude the possibility that l might be clobbered by actions in the decompositions of other tasks. In our framework, only primitive tasks can change the state of the world, non-primitive tasks are not allowed to have direct effects. Instead, we express high-level effects as constraints of the form (n, l) so that the planner verifies those effects to hold. Thus we avoid the previous problem, but we still benefit from guiding search with high level effects (one of the primary reasons they are often used in implemented planning systems).

Conditions

HTN planners often allow several types of conditions in methods. For instance NONLIN has *use-when, supervised,* and *unsupervised* conditions.

Supervised conditions, similar to preconditions in STRIPS-style planning, are those that the planner needs to accomplish, thus in our framework, they appear as goal nodes in task networks. In the task network shown in Fig. 5, the conditions $\mathbf{clear(v_1)}$ and $\mathbf{clear(v_2)}$ appear as goal tasks for that reason.

Unsupervised conditions are conditions that are needed for achieving a task but supposed to be accomplished by some other part of the task network (or the initial state). For example, a **load(package,vehicle)** task in a transport logistics domain would require the package and vehicle to be at the same location, but it might be the responsibility of another task (e.g. a vehicle dispatcher) to accomplish that condition. Thus, the load task must only verify the condition to be true and it must not try to achieve it by task decompositions, or insertions of new actions. In our framework, we represent unsupervised conditions as state constraints so that the planner tries to find variable bindings/task orderings that would make them true, but it does not plan for them by inserting new actions or doing task decompositions. NONLIN ignores the unsupervised conditions until all tasks are expanded into primitive tasks, which is not always an efficient strategy. UMCP, on the other hand, tries to process unsupervised conditions at higher levels to prune the search space.

HTN planners employ filter conditions (called use-when conditions in NONLIN) for deciding which methods to try for a task expansion and reduce the branching factor by eliminating irrelevant methods. For example consider the task of going to New York, and the method of accomplishing it by driving. One condition necessary for this method to succeed is having a driver's license. Although a driver's license can be obtained by learning how to drive and going through the paperwork, the user of the planner might consider this unacceptable, and in that case he would specify having a driver's license not as a goal task but as a filter condition , and the method of driving to New York would not be considered if the agent does not have a driver's license at the appropriate point in the plan. In (Collins *et al.*, 1992), Collins and Pryor state that filter conditions are ineffective. They argue that filter conditions do not help pruning the search space for partial order planners, because it is not possible to check whether they hold or not in an incomplete plan. They also empirically demonstrate that ignoring the filter conditions until all the subgoals are achieved is quite inefficient. Although their study of filter conditions is in the context of STRIPS representation, to a large extend it also applies to HTN planning. NONLIN, for instance, evaluates filter conditions as soon as they are encountered, and unless it can establish those conditions to be necessarily true, it will backtrack. Thus, NONLIN sometimes backtracks over filter conditions which would have been achieved by actions in later task expansions. Hence NONLIN is not complete. Although it might not always be possible to determine whether a filter condition is true in an incomplete plan, filter conditions can still be used to prune the search space. Our framework represents filter conditions as state constraints and planning algorithms based on this framework can employ constraint satisfaction/propagation techniques to prune inconsistent task networks. For example if a task network contains the filter condition (l,n), and also another constraint $(n_1, \neg l, n_2)$, one can deduce that n should be either before n_1, or after n_2. Furthermore, some filter conditions might not be affected by the actions (e.g. conditions on the type of objects), and thus it suffices to check the initial state to evaluate those. This kind of filter conditions can also be very helpful in pruning the search space.

Overall, our constraint language provides a principled way of representing many kinds of conditions, and our planner UMCP employs techniques for using them effectively without sacrificing soundness or completeness.

Constraints and Critics

One attractive feature of HTN systems is that by using various sorts of critics, they can handle many different kinds of interactions, thus allowing the analysis of potential problems in plans, and preventing later failure and backtracking. However, this mechanism has been little explored in the formal literature, primarily due to the procedural nature of handling interactions between subtasks. [5] In our framework, we represent interactions using constraints: temporal constraints of the form $(n \prec n')$ denote temporal interactions, and state constraints of the form (n, l, n') denote deleted condition interactions. We have provided a conservative set of constraints in section 3.1 to keep the paper in focus. This set can easily be extended to express other kinds of interactions (for example the resource interactions of (Wilkins, 1988b)).

HTN planners typically use their critics for guiding the search at higher levels, before all subtasks have been reduced to primitive tasks. In our formalism, this job is performed by the critic function τ. τ inputs an initial state I, and a task network d, and outputs a set of task networks Γ. τ manipulates the constraints and the tasks in d. For example, if τ encounters a constraint (n_1, l, n_2), and a primitive task(labeled with n_3) that asserts $\neg l$, it might augment the constraint formula to contain $(n_3 \prec n_1) \vee (n_2 \prec n_3)$. Alternatively, τ might decide that it is time to commit to an ordering, and output two task networks, one with $(n_3 \prec n_1)$, and another with $(n_2 \prec n_3)$.

Another usage of constraints is encoding control information. When the user encodes a domain, it might be known in advance that certain variable bindings, task orderings etc. lead to dead ends. These can be eliminated by posting constraints (in methods) so that the planner does not go waste time deriving this information, thus gaining efficiency. (This ability to encode known shortcuts and/or pitfalls was offered as a major motivation for the move to procedural networks in NOAH (Sacerdoti, 1977)).

5 A Hierarchical Planning Procedure.

[5]The obvious exception is deleted precondition interactions, which have been analyzed *ad nauseum*.

procedure UMCP:
1. Input a planning problem $\mathbf{P} = \langle d, I, \mathcal{D} \rangle$.
2. if d is primitive, then
   If $comp(d, I, \mathcal{D}) \neq \emptyset$, return a member of it.
   Otherwise return FAILURE.
3. Pick a non-primitive task node $(n : \alpha)$ in d.
4. Nondeterministically choose a method m for α.
5. Set $d := reduce(d, n, m)$.
6. Set $\Gamma := \tau(d, I, \mathcal{D})$.
7. Nondeterministically set $d :=$ some element of Γ.
8. Go to step 2.

Figure 7: **UMCP**: Universal Method-Composition Planner

Using the syntax and semantics developed in the previous section, we can now formalize the HTN planning procedure that we presented in Figure 3. Figure 7 presents our formalization, which we call **UMCP** (for Universal Method-Composition Planner).

It should be clear that **UMCP** mimics the definition of $sol(d, I, \mathcal{D})$, except for Steps 6 and 7 (which correspond to the critics). As discussed before, HTN planners typically use their critics for detecting and resolving interactions among tasks (expressed as constraints) in task networks at higher levels, before all subtasks have been reduced to primitive tasks. By eliminating some task orderings and variable bindings that lead to dead ends, critics help prune the search space. In our formalism, this job is performed by the critic function τ. τ takes as input an initial state I, a task network d, and a planning domain $\mathcal{D}$; and produces as its output a set of task networks Γ. Each member of Γ is a candidate for resolving some[6] of the conflicts in d. We need to put two restrictions on τ:

1. If $d' \in \tau(d, I, \mathcal{D})$ then $sol(d', I, \mathcal{D}) \subseteq sol(d, I, \mathcal{D})$. Thus, any plan for d' must be a plan for d ensuring soundness.

2. If $\sigma \in sol_k(d, I, \mathcal{D})$ for some k, then there exists $d' \in \tau(d, I, \mathcal{D})$ such that $\sigma \in sol_k(d', I, \mathcal{D})$.

   Thus, whenever there is a plan for d, there is a plan for some member d' of $\tau(d, I, \mathcal{D})$. In addition, if the solution for d is no further than k expansions, so is the solution

[6] It might be impossible or too costly to resolve some conflicts at a given level, and thus handling those conflicts can be postponed.

for d'. The latter condition ensures that τ does not create infinite loops by undoing previous expansions.

In contrast to the abundance of well understood STRIPS-style planning algorithms (such as (Fikes *et al.*, 1971; Chapman, 1987; Barett *et al.*, 1992; Kambhampati, 1992)), HTN planning algorithms have typically not been proven to be sound or complete. However, using the formalism in this paper, we can establish the soundness and completeness of the HTN planning algorithm **UMCP**.

Theorem 2 (Soundness) *Whenever* **UMCP** *returns a plan, it achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.*

Theorem 3 (Completeness) *Whenever* **UMCP** *fails to find a plan, there is no plan that achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.*

These results follow directly from the equivalence theorem using the fact that **UMCP** directly mimics $sol()$. The restrictions on the critic function ensure that τ does not introduce invalid solutions and that it does not eliminate valid solutions. The details of the proofs are in the appendix.

6 Expressivity

It has long been a topic of debate whether HTN planning is merely an "efficiency hack" over STRIPS-style planning, or HTN planning is actually more expressive than STRIPS-style planning. Our HTN framework enables us to address this question formally.

There is not a well established definition of expressivity for planning languages. It is possible to define expressivity based on model-theoretic semantics, operational semantics, and even on the computational complexity of problems that can be represented in the planning language. We have devised formal definitions for each of these cases, and proved that HTN planning is strictly more expressive than STRIPS-style planning according to all three of them.

6.1 Model-Theoretic Expressivity

Baader (Baader, 1990) has presented a definition of expressivity for knowledge representation languages. Below we describe Baader's definition of expressivity and adapt it to planning languages.

Definition of Model-Theoretic Expressivity

Baader's notion of expressivity is based on the idea that if a language L_1 can be expressed by another language L_2, then for any set of sentences Γ_1 in L_1, there must be a corresponding set of sentences Γ_2 in L_2 with the same meaning. Baader captures "the same meaning" by requiring the two set of sentences have the same set of models. Since it is possible that L_2 can contain symbols not necessary for expressing L_1, and the name of the symbols used must not make a difference, it suffices for the set of models for Γ_1 and the set of models for Γ_2 be equivalent modulo a symbol translation function ω.

More formally, given a function ω from the set of symbols in L_1 into the set of symbols in L_2, two models M_1 and M_2 are defined to be equivalent module ω, denoted as $M_1 =^{\omega} M_2$ iff for any symbol s in L_1, $M_1(s) = M_2(\omega(s))$, in other words, s and $\omega(s)$ must have the same interpretation.

The equivalence between models can be extended to sets of models as follows: Two sets of models $\mathcal{M}_1$, and $\mathcal{M}_2$ are equivalent modulo a symbol translation function ω, denoted by $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$, iff for any model $M_1 \in \mathcal{M}_1$, there exists a model $M_2 \in \mathcal{M}_2$ such that $M_1 =^{\omega} M_2$, and for any model $M_2 \in \mathcal{M}_2$, there exists a model $M_1 \in \mathcal{M}_1$ such that $M_1 =^{\omega} M_2$.

A knowledge representation language L_1 can be expressed by another knowledge representation language L_2, iff there exists a function ψ from the set of sentences in L_1 to the set of sentences in L_2, and a function ω from the set of symbols in L_1 to the set of symbols in L_2 such that for any set of sentences Γ from L_1, the set of models $\mathcal{M}_1$ satisfying Γ and the set of models $\mathcal{M}_2$ satisfying $\psi(\Gamma)$ are equivalent modulo ω, i.e. $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$.

Although the internal structures of the models for planning languages and knowledge representation languages are different, we can still use the same definition of expressivity, by providing a definition for equivalence of HTN models:

Given two HTN models M_1, M_2 for two HTN languages $\mathcal{L}_1, \mathcal{L}_2$, and a symbol translation function ω from $\mathcal{L}_1$ to $\mathcal{L}_2$, $M_1 =^{\omega} M_2$ iff

1. For any ground primitive task α, any state s, and any ground literal l in $\mathcal{L}_1$, l is true in $\mathcal{F}_{M1}(\alpha, s)$ iff $\omega(l)$ is true in $\mathcal{F}_{M2}(\omega(\alpha), \omega(s))$.
2. For any ground non-primitive task α in $\mathcal{L}_1$, $\omega(\mathcal{T}_{M1}(\alpha))$ covers $\mathcal{T}_{M2}(\omega(\alpha))$, and $\mathcal{T}_{M2}(\omega(\alpha))$ covers $\omega(\mathcal{T}_{M1}(\alpha))$.

Recall that "covers" was defined in Section 3.2.

Now we can formally define expressivity of planning languages. A planning language$\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, iff the following three conditions hold:

1. There exists a symbol translation function ω from the set of symbols in $\mathcal{L}_1$ to the set of symbols in $\mathcal{L}_2$,

2. There exists a function ψ from the set of sentences in $\mathcal{L}_1$ to the set of sentences in $\mathcal{L}_2$,

3. For any set of sentences Γ_1 in $\mathcal{L}_1$, whenever $\mathcal{M}_1$ is the set of all models that satisfy Γ_1 and $\mathcal{M}_2$ is the set of all models that satisfy $\psi(\Gamma_2)$, it is the case that $\mathcal{M}_1 =^{\omega} \mathcal{M}_2$.

When a planning language $\mathcal{L}_1$ can be expressed by a planning language $\mathcal{L}_2$, but not vice versa, then $\mathcal{L}_2$ is strictly more expressive than $\mathcal{L}_1$.

Before proceeding to compare the expressive power of HTN planning and STRIPS-style planning, we need to present a formal semantics for STRIPS-style planning that is compatible with the semantics for HTN planning.

Semantics for STRIPS

A semantic structure for STRIPS-style planning has the same form as a semantic structure for HTN planning, with some restrictions. Thus it is a triple $M = \langle \mathcal{S}_M, \mathcal{F}_M, \mathcal{T}_M \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{F}$ interprets actions as state transitions, and $\mathcal{T}$ interprets non-primitive tasks as sets of ground primitive task networks, with the following restrictions:

1. Since STRIPS representation lacks the notion of compound tasks, non-primitive tasks consist of only goal tasks, and $\mathcal{T}$ is not defined for compound tasks.

2. In STRIPS-style planning, any executable sequence of actions that make the goals true is a valid plan, thus, given any goal task $achieve[l]$, $\mathcal{T}$ maps $achieve[l]$ to the set of all ground sequences of actions, each with the (implicit) constraint that l is true in the final state.

HTNs versus STRIPS

It is fairly easy to show that the STRIPS language can be expressed by the HTN planning language. All we need to is to present two functions ψ and ω and show that the translation preserves the set of models.

Since HTN planning does not require any extra symbols for expressing STRIPS-style planning, ω is defined to be the identity function. Thus the HTN representation is going to use exactly the same set of constants, predicates, and actions (primitive tasks) as its STRIPS counterpart.

Given a set of STRIPS-style operators Γ, here is how we define $\psi(\Gamma)$:

- $\psi(\Gamma)$ contains exactly the same set of operators as Γ.

- For each goal task $achieve[l]$ and each action f, $\psi(\Gamma)$ contains the method

$$(achieve[l], [(n_1 : do[f])(n_2 : achieve[l]), (n_1 \prec n_2)])$$

Since there is an implicit method for each goal task stating it can be expanded to a dummy task when the goal literal is already true, with the methods in $\psi(\Gamma)$, any goal task can be expanded to any sequence of actions, and such a sequence of actions would be a plan for the goal task whenever all the actions are executable and the goal literal is true in the end of the plan. Thus, the methods precisely reflect the restrictions on the models of STRIPS-style planning, and as a result Γ and $\psi(\Gamma)$ have exactly the same set of models. Thus we conclude

Lemma 1 *The* STRIPS *language can be expressed by the* HTN *language with respect to model-theoretic expressivity.*

The converse is not true. To prove that the STRIPS language is not as expressive as the HTN language, we are going to construct an HTN planning domain Γ, and show that there does not exist any STRIPS-style planning domain (i.e. set of STRIPS operators) with an equivalent set of models.

Note that the set of plans for any STRIPS-style planning domain always forms a regular set: One can define a finite automata with the same states as the STRIPS domain, with state transitions corresponding to the actions, and the goal states in the STRIPS domain would be designated as the final states in the automata. On the other hand, the set of plans for an HTN planning domain can be any arbitrary context-free set, including those context-free sets that are not regular. Given a context-free grammar, we can declare one compound task symbol for each non-terminal of the grammar, one primitive task symbol for each terminal of the grammar, and for each grammar rule of the form $X \rightarrow YZ$, we can declare a method $(\alpha_X \;\; [(n_1 : \alpha_Y)(n_2 : \alpha_Z)\; (n_1 \prec n_2)])$.

Given an HTN planning domain Γ that corresponds to a context-free but not regular grammar, Γ will have a minimum model[7] M such that $\mathcal{T}_M$ will map compound task symbols (which correspond to the non-terminal symbols of the grammar) to sets of totally ordered primitive task networks (or equivalently, to context-free sets of strings from the terminal symbols of the grammar). Since $\mathcal{T}_M$ maps compound tasks into context-free but not regular sets, no STRIPS-style planning domain can have a model equivalent to M. Thus we state the following lemma:

Lemma 2 *The* HTN *language cannot be expressed by the* STRIPS *language with respect to model-theoretic expressivity.*

[7]refer to the proof of Theorem 1 to see how to construct a minimum model

Thus, from Lemma 1 and Lemma 2 we can conclude the following:

Theorem 4 *The* HTN *language is strictly more expressive than the* STRIPS *language with respect to model-theoretic expressivity.*

6.2 Alternative Definitions of Expressivity

There are other ways of defining expressivity. For instance, we can modify the third condition in the definition for the expressivity of planning languages so that it reads as follows:

- For any set of sentences Γ_1 in $\mathcal{L}_1$, whenever M_1 is the minimum model that satisfies Γ_1, and M_2 is the minimum model that satisfies $\psi(\Gamma_1)$, it is the case that $M_1 =^{\omega} M_2$.

From the operational semantics point of view, this corresponds to requiring that for every planning problem expressed in L_1, there exists a corresponding planning problem expressed in L_2 and the set of solutions for both these problems are equivalent. Although this definition of expressivity based on operational semantics is quite different from the definition of model-theoretic expressivity, it yields the same result: Since the set of solutions to an HTN planning problem can be any context-free set, whereas the set of solutions to a STRIPS-style planning problem always is a regular set, a proof similar to that of Lemma 2 can be constructed to prove that HTN planning is strictly more expressive according to this definition of expressivity. Thus, we can conclude the following:

Theorem 5 *The* HTN *language is strictly more expressive than the* STRIPS *language with respect to operational expressivity.*

The details of the definition of operational expressivity and the proof can be found in (Erol *et al.*, 1994b).

Note that in previous definitions of expressivity, ψ is not restricted to be computable in polynomial time (or even to be computable at all!), nor are there any restrictions on the size of $\psi(\Gamma)$ in terms of the size of Γ. In defining expressivity, one can discard the equivalence of models and instead ask whether there exists a polynomial (or computable) transformation τ from the set of planning problems that can be represented in $\mathcal{L}_1$ to the set of planning problems that can be represented in $\mathcal{L}_2$ such that any planning problem $\mathbf{P}$ in $\mathcal{L}_1$ has a solution iff $\tau(\mathbf{P})$ also has a solution. This definition of expressivity for a planning language is based on the computational complexity of telling which planning problems represented in that language have solutions. In (Erol *et al.*, 1994b) we show that the complexity of HTN planning is strictly semi-decidable, whereas in (Erol *et al.*, 1992a), we show that the complexity of STRIPS-style planning is much easier, more specifically, EXPSPACE-complete. This clearly shows that there does not exist any computable transformation from HTN planning to STRIPS-style planning. Hence we can state the following theorem:

Theorem 6 *The* HTN *language is strictly more expressive than the* STRIPS *language with respect to complexity-based expressivity.*

The details of the definition of complexity-based expressivity, together with the proofs that HTN planning is more expressive with respect to this definition can be found in (Erol *et al.*, 1994b).

7 Conclusions

One big obstacle to understanding the nature of hierarchical task network (HTN) planning has been the lack of a clear theoretical framework. In this paper, we have presented a formal syntax and semantics for HTN planning. Based on this syntax and semantics, we have defined an algorithm for HTN planning, and have proved it to be sound and complete.

This formalism also enables us to do complexity analyses of HTN planning, to assess the expressive power of the use of HTNs in planning, and to compare HTNs to planning with STRIPS-style operators. For example, we have been able to prove that HTN planning, as defined here, is formally more expressive than planning without decompositions (Erol *et al.*, 1994c; Erol *et al.*, 1994b). We are working on a deeper complexity analysis of HTNs and towards an understanding of where the complexity lies.

Our semantics characterizes various features of HTN planning systems, such as tasks, task networks, filter conditions, task decomposition, and critics. We believe that this more formal understanding of these aspects of planning will make it easier to encode planning domains as HTNs and to analyze HTN planners. Furthermore, the definition for whether a given model satisfies a planning domain can provide a criterion for telling whether a given set of methods and operators correctly describe a particular planning domain. We are currently exploring these further.

Finally, we are starting to explore the order in which tasks should be expanded to get the best performance, and more generally, in which order all commitments (variable bindings, temporal orderings, choice of methods) should be made. This will involve both algorithmic and empirical studies. Our long term goals are to characterize planning domains for which HTN planning systems are suitable, and to develop efficient planners for those domains. Our framework provides the necessary foundation for such work.

References

(Allen *et al.*, 1990) Allen, J.; Hendler, J. and Tate, A. editors. *Readings in Planning.* Morgan-Kaufmann, San Mateo, CA, 1990.

(Baader, 1990) Baader, F. A formal definition for expressive power of knowledge representation languages. In *Proceedings of the 9th European Conference on Artificial Intelligence*, Stockholm, Sweden, Aug. 1990. Pitman.

(Barett *et al.*, 1992) Barrett, A. and Weld, D. Partial Order Planning. Technical report 92-05-01, Computer Science Dept., University of Washington, June, 1992.

(Chapman, 1987) Chapman, D. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

(Collins *et al.*, 1992) Gregg Collins and Louise Pryor. Achieving the functionality of filter conditions in a partial order planner In *Proc. AAAI-92*, 1992, pp. 375—380.

(Drummond, 1985) Drummond, M. Refining and Extending the Procedural Net. In *Proc. IJCAI-85*, 1985.

(Erol *et al.*, 1992a) Erol, K.; Nau, D.; and Subrahmanian, V. S. On the Complexity of Domain Independent Planning. In *Proc. AAAI-92*, 1992, pp. 381—387.

(Erol *et al.*, 1992b) Erol, K.; Nau, D.; and Subrahmanian, V. S. When is planning decidable? In *Proc. First Internat. Conf. AI Planning Systems*, pp. 222–227, June 1992.

(Erol *et al.*, 1994b) Erol, K.; Hendler, J.; and Nau, D. Complexity results for hierarchical task-network planning. To appear in *Annals of Mathematics and Artificial Intelligence* Also available as Technical report CS-TR-3240, UMIACS-TR-94-32, ISR-TR-95-10, Computer Science Dept., University of Maryland, March 1994.

(Erol *et al.*, 1994c) Erol, K.; Hendler, J.; and Nau, D. HTN Planning: Complexity and Expressivity. To appear in *Proc. AAAI-94*, 1994.

(Fikes *et al.*, 1971) Fikes, R. E. and Nilsson, N. J. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

(Kambhampati *et al.*, 1992) Kambhampati, S. and Hendler, J. "A Validation Structure Based Theory of Plan Modification and Reuse" *Artificial Intelligence*, May, 1992.

(Kambhampati, 1992) Kambhampati, S. "On the utility of systematicity: understanding trade-offs between redundancy and commitment in partial-ordering planning," unpublished manuscript, Dec., 1992.

(Lansky, 1988) Lansky, A.L. Localized Event-Based Reasoning for Multiagent Domains. *Computational Intelligence Journal*, 1988.

(Mcallester *et al.*, 1991) Mcallester, D. and Rosenblitt, D. Systematic nonlinear planning. In *Proc. AAAI-91*, 1991.

(Minton *et al.*, 1991) Minton, S.; Bresna, J. and Drummond, M. Commitment strategies in planning. In *Proc. IJCAI-91*, 1991.

(Nilsson, 1980) Nilsson, N. *Principles of Artificial Intelligence,* Morgan-Kaufmann, CA. 1980.

(Penberthy *et al.*, 1992) Penberthy, J. and Weld, D. S. UCPOP: A Sound, Complete, Partial Order Planner for ADL *Proceedings of the Third International Conference on Knowledge Representation and Reasoning, October 1992*

(Sacerdoti, 1977) Sacerdoti, E. D. *A Structure for Plans and Behavior,* Elsevier-North Holland. 1977.

(Tate *et al.*, 1990) Tate, A.; Hendler, J. and Drummond, D. AI planning: Systems and techniques. *AI Magazine*, (UMIACS-TR-90-21, CS-TR-2408):61–77, Summer 1990.

(Tate, 1990) Tate, A. Generating Project Networks In *Proc. IJCAI-77*, 1977. pp. 888–893.

(Vere, 1983) Vere, S. A. Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):246–247, 1983.

(Wilkins, 1988a) Wilkins, D. Domain-independent Planning: Representation and Plan Generation. In Allen, James; Hendler, James; and Tate, Austin, editors 1990, *Readings in Planning.* Morgan Kaufman. 319—335.

(Wilkins, 1988b) Wilkins, D. *Practical Planning: Extending the classical AI planning paradigm,* Morgan-Kaufmann, CA. 1988.

(Yang, 1990) Yang, Q. Formalizing planning knowledge for hierarchical planning *Computational Intelligence* Vol.6., 12–24, 1990.

A Appendix

Theorem 1 (Equivalence Theorem) Given a task network d,an initial state I, and a plan σ, σ is in $sol(d, I, \mathcal{D})$ iff any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$.

Proof. $\rightarrow$. Since $sol(d, I, \mathcal{D})$ is defined recursively in terms of completions and reductions, it suffices to show that (a)Given any primitive task-network d, if $\sigma \in comp(d, I, \mathcal{D})$, then any model that satisfy $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$, (b) for any model M that satisfies $\mathcal{D}$, if M satisfies $solves(\sigma, d', I)$, and $d' \in red(d, I, \mathcal{D})$, it also satisfies $solves(\sigma, d, I)$.

(a) Assume $\sigma \in comp(d, I, \mathcal{D})$. Then, σ is a totally ordered ground instance of d that makes the constraint formula true. Let M be any model that satisfies $\mathcal{D}$. Because M satisfies the operators, $\mathcal{F}_M$ will project the intermediate states to be exactly the same as projected in the completion. Furthermore, the constraint formula is evaluated in the same way both for finding completions and for determining whether a model satisfies $solves(\sigma, d, I)$. Thus, given a primitive task network d, and a model M that satisfies $\mathcal{D}$, M satisfies $solves(\sigma, d, I)$ iff $\sigma \in comp(d, I, \mathcal{D})$.

(b) Let d' be in $red(d, I, \mathcal{D})$. Then there exists a method m, and a task node n in d such that $d' = reduce(d, n, m)$. Let M be a model that satisfies $\mathcal{D}$, and also $solves(\sigma, d', I)$.

Without loss of generality, let's assume d, m, d' have the following forms:

$$\begin{array}{lll} d = & & [(n_1 : \alpha_1) \ldots (n_k : \alpha_k) \;\; (n : \alpha), \phi], \\ m = & (\alpha, & [(n'_1 : \alpha'_1) \ldots (n'_j : \alpha'_j), \psi]), \\ d' = & & [(n'_1 : \alpha'_1) \ldots (n'_j : \alpha'_j) \;\; (n_1 : \alpha_1) \ldots (n_k : \alpha_k), \phi' \wedge \psi]. \end{array}$$

Since M satisfies $solves(\sigma, d', I)$, there exists
$d'_1 \in \mathcal{T}(\alpha'_1), \ldots, d'_j \in \mathcal{T}(\alpha'_j), d_1 \in \mathcal{T}(\alpha_1), \ldots, d_k \in \mathcal{T}(\alpha_k)$ such that σ is a plan for $compose(d_1, \ldots, d'_j, \; d_1, \ldots, d_k, \phi' \wedge \psi)$.

Thus, there exists a matching π between σ and $compose(d'_1, \ldots, d'_j, \; d_1, \ldots, d_k, \phi' \wedge \psi)$, such that the constraint formula of $compose(d'_1, \ldots, d'_j, \psi)$, which correspond to the portion of the task network corresponding to the expansion of α, is satisfied. From this fact and that M satisfies the method m, we conclude there exists a $d'' \in \mathcal{T}(\alpha)$ and a matching π' such that σ makes the constraint formula of d'' true.

Consider $compose(d_1, \ldots, d_k, \; d'', \phi) \in \mathcal{T}(d)$. Construct a matching π'' by extending π' to $d_1, \ldots, d_k$ (taking the same value as π for those places). σ satisfies ϕ and the constraints of $d_1, \ldots, d_k, \; d''$. Thus M satisfies $solves(\sigma, d, I)$.

$\leftarrow$. We will show that whenever $\sigma \notin sol(d, I, \mathcal{D})$, there exists a model M that satisfies $\mathcal{D}$, but not $solves(\sigma, d, I)$.

Here is how we construct $M =< \mathcal{S}, \mathcal{F}, \mathcal{T} >$:

- $\mathcal{S} = 2^{\{ground\ atoms\}}$.

- $\mathcal{F}(f, c_1 \ldots, c_k, s) = (s - N\theta) \cup P\theta$, whenever the operator for f is of the form $(f(v_1, \ldots, v_k), l_1, \ldots, l_k)$, where θ is the substitution $\{c_i/v_i | 1 \leq i \leq k\}$, and N, P are the sets of negative and positive literals in $\{l_1, \ldots, l_k\}$, respectively.

- $\mathcal{T}(\alpha) = \{d | d$ is a ground instance of a primitive task-network obtained from α by a finite number of reductions$\}$, for any non-primitive task α.

When $\mathcal{T}$ is extended to cover task networks as defined in Section 3.2, we observe that $\mathcal{T}(d) = \{d' | d'$is a ground instance of a primitive task-network obtained from d by a finite number of reductions$\}$ for any task network d.

$\mathcal{F}$ is defined such that M satisfies all the operators in $\mathcal{D}$. By definition of $\mathcal{T}$, whenever $d' \in red(d, I, \mathcal{D})$, $\mathcal{T}(d') \subseteq \mathcal{T}(d)$. Thus M also satisfies all the methods in $\mathcal{D}$.

Given a primitive task network d, and a model M that satisfies $\mathcal{D}$, M satisfies $solves(\sigma, d, I)$ iff $\sigma \in comp(d, I, \mathcal{D})$

Let d be a primitive task network such that $\sigma \notin sol(d, I, \mathcal{D})$. Then $\sigma \notin comp(d, I, \mathcal{D})$, either. In part (a) of the proof, we showed that for any model that satisfies the operators, $comp(d, I, \mathcal{D})$ is exactly the set of plans that solves the primitive task network d. Thus we conclude M does not satisfy $solves(\sigma, d, I)$.

Consider the alternative where d is a non-primitive task network such that $\sigma \notin sol(d, I, \mathcal{D})$. Assume M satisfies $solves(\sigma, d, I)$. Then there exists a primitive task network $d' \in \mathcal{T}(d)$ such that σ is a plan for d'. From part (a) of the proof, σ must be in $comp(d', I, \mathcal{D})$. However, since $\mathcal{T}(d)$ contains only primitive task networks that can be obtained by a finitenumber of reductions from d, we conclude $\sigma \in sol(d, I, \mathcal{D})$, which is a contradiction. ■

Theorem 2 (Soundness) Whenever **UMCP** returns a plan, it achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.

Proof. Let's assume on input $\mathbf{P} =< d, I, \mathcal{D} >$, **UMCP** halts in n iterations. and returns σ. Using induction on n, we prove that $\sigma \in sol(d, I, \mathcal{D})$, and from the equivalance theorem we conclude that any model that satisfies $\mathcal{D}$ also satisfies $solves(\sigma, d, I)$.

[Base case: $n = 0$.] d must be a primitive task network and $\sigma \in comp((d, I, \mathcal{D})$. Thus, $\sigma \in sol(d, I, \mathcal{D})$.

[Induction Hypothesis] Assume for $n < k$ if **UMCP** returns σ in n iterations, then $\sigma \in sol(d, I, \mathcal{D})$.

Suppose on input $\mathbf{P} =< d, I, \mathcal{D} >$ **UMCP** halts in $n = k$ iterations. and returns σ. Then d is a non-primitive task network. Let $d_1 = reduce(d, n, m)$ be the value assigned to d at step 5 of **UMCP**in the first iteration, and let $d_2 \in \tau(d_1, I, \mathcal{D})$ be the value assigned to d at step 7 of **UMCP**in the first iteration.

On input $\mathbf{P} =< d_2, I, \mathcal{D} >$, the planner halts in $k-1$ steps and returns σ. Thus, by induction hypothesis, $\sigma \in sol(d_2, I, \mathcal{D})$. From restriction 1 on the critic function $\tau()$ and from $d_2 \in \tau(d_1, I, \mathcal{D})$ we conclude $\sigma \in sol(d_1, I, \mathcal{D})$. Since $d_1 \in red(d, I, calD)$, from the definition of $sol()$ we conclude $\sigma \in sol(d, I, \mathcal{D})$. ■

Theorem 3 (Completeness) Whenever **UMCP** fails to find a plan, there is no plan that achieves the input task network at the initial state with respect to all the models that satisfy the methods and the operators.

Proof. Assume $\sigma \in sol(d, I, \mathcal{D})$. Let k be the minimum number of reductions needed to derive σ; i.e. $\sigma \in sol_{k+1}(d, I, \mathcal{D})$, but $\sigma \notin sol_k(d, I, \mathcal{D})$. We show that there exists a sequence of non-deterministic choices such that the **UMCP** halts in k iterations and returns a plan.

Proof by induction on k.

[Base case: $k = 0$.] In that case $\sigma \in comp(d, I, \mathcal{D})$, and **UMCP** finds a plan in in step 2.

[Induction Hypothesis] Assume whenever the number of reductions needed to derive σ is less then j (i.e. $k < j$), there exists a sequence of non-deterministic choices for which **UMCP** returns a plan in k iterations.

Let $\sigma \in sol((d, I, \mathcal{D})$, and it takes exactly j reductions to derive σ. Let $(n : \alpha)$ be the node picked by **UMCP** in step 3 in the first iteration (Since this is not a non-deterministic choice, the planner could pick any non-primitive node at that step.) Let m be the method used for reducing $(n : \alpha)$ in the derivation. Let $d_1 = reduce(d, n, m)$. $reduce()$ is defined in Section 3.2 such that the order of reductions does not matter; i.e. For a task network d that contains two non-primitive tasks n_1, n_2 with corresponding methods m_1, m_2,

$$reduce(reduce(d, n_1, m_1), n_2, m_2) \quad and \quad reduce(reduce(d, n_2, m_2), n_1, m_1)$$

are equal modulo variable and node label names. Thus, since we obtained d_1 from d by reducing n with the same method used in the derivation, we conclude $\sigma \in sol_{j-1}((d_1, I, \mathcal{D})$.

As $\sigma \in sol_{j-1}((d_1, I, \mathcal{D})$, by the second restriction on $\tau()$, there exists $d_2 \in \tau(d_1, I, \mathcal{D})$ such that $\sigma \in sol_{j-1}((d_2, I, \mathcal{D})$. Let the planner non-deterministically choose d_2 to assign to d at step 7 in the first iteration. By induction hypothesis, the planner will halt in $j-1$ more iterations and return a plan. ■