Planning in Answer Set Programming using Ordered Task Decomposition

Jürgen Dix[1]*, Ugur Kuter, and Dana Nau[2]

[1] The University of Manchester
Dept. of CS, Oxford Road
Manchester M13 9PL, UK
dix@cs.man.ac.uk
http://www.cs.man.ac.uk/~jdix
[2] University of Maryland
Dept. of CS, A.V. Williams Building
College Park, MD 20752, USA
{ukuter,nau}@cs.umd.edu

Abstract. In this paper we introduce a formalism for solving *Hierarchical Task Network* (HTN) Planning using *Answer Set Programming* (ASP). We consider the formulation of HTN planning as described in the *SHOP* planning system and define a systematic translation method from *SHOP*'s representation of the planning problem into logic programs with negation. We show that our translation is *sound* and *complete*: answer sets of the logic program obtained by our translation correspond exactly to the solutions of the planning problem. We compare our method to (1) similar approaches based on non-HTN planning and (2) *SHOP*, a dedicated planning system. We show that our approach outperforms non-HTN methods and that its performance is better with ASP systems that allow for nonground programs than with ASP systems that require ground programs.
Keywords: HTN planning, nonmonotonic reasoning, ASP systems, benchmarks

1 Introduction

In the past few years, the availability of very fast nonmonotonic systems based on logic programming (LP) made it possible to attack problems from other, non-LP areas, by translating these problems into logic programs and running a fast prover on them. One of the first such system was *smodels* [1] and one of the early applications [2] was to transform planning problems in a suitable way and to run *smodels* on them (see also [3]).

Since then more implemented systems with different properties for dealing with logic programs have become available: *DLV* [4], *XSB* [5, 6] to cite the most well-known. In addition, the paradigm of Answer Set Programming (ASP)

* This paper is an extended abstract of a paper that is currently submitted as a regular paper for *Theory and Practice of Logic Programming*.

emerged [7] is based on the following two key ideas: (1) solving problems by computing models for logic programs rather than by evaluating queries against logic programs (as used to be done in conventional logic programming), (2) addressing the problems located on the second level of the polynomial hierarchy which seem to be well suited to be tackled with the machinery of answer sets. In particular many planning problems fit in this picture. Indeed, the problems on the first two levels of polynomial time hierarchy are covered by the current ASP implementations.

In this paper, we investigate the ways of formulating and solving Hierarchical Task Network (HTN) planning problems using nonmonotonic logic programs under the ASP semantics. HTN planning [8–11] is an AI-planning paradigm in which the goals of the planner are defined in terms of activities (tasks) and the planning process is accomplished by using the techniques of task decomposition. There are several well-known HTN planning systems such as *Universal Method Composition Planner* (*UMCP*) [9], *Simple Hierarchical Ordered Planner* (*SHOP*) [11], and *SHOP2* (a total-order planner with partially ordered subtasks) [12]. In this work, we focus on the *SHOP* planning system, which is a domain-independent HTN planning system that is built around the concept called *ordered task decomposition*.

We describe a systematic translation method $\mathfrak{Trans}(\cdot)$ which transforms HTN-planning problems as formalised in *SHOP* into logic programs with negation. Our basic goal is that an appropriate semantics of the logic program should correspond to the solutions (plans) of the planning problem. We have adapted the syntax of the *smodels* software for our transformation, although we are also experimenting with other systems like *DLV* and *XSB*.

Our experimental results suggest that both (1) encodings using HTN planning are better than other encodings, because the HTN control knowledge can be used very naturally to prune irrelevant branches of the search space; and (2) running an ASP system on non-ground programs (obtained from planning problems) results in a drastic performance relative to *smodels*, thus bringing our method closer to dedicated planning systems like *SHOP*.

This paper is organised as follows. In the following section, we present the related work. In Section 2, we describe the basic HTN Planning concepts as they are defined in *SHOP*. In Section 3, we first present our causal theory for HTN planning and then our translation methodology to transform HTN-planning problems into logic programs with negation. Section 4 contains our theoretical and experimental results. Our main theorem states that our translation method is correct and complete with respect to *SHOP*. Finally, we conclude with Section 5 and provide our future research directions.

1.1 Related Work

There are many efforts in the literature for formulating actions in logic programs and solving planning problems by using formulations such as [13–17]. The idea in all these works is that representing a given computational problem by a logic

program whose models correspond to the solutions for the original problem. This idea was the main inspiration for our work presented here.

[18] proposes a declarative language, called the K language, for planning with incomplete information. The K language makes it possible to describe transitions between knowledge states, which may not be complete, regarding the world. This language is implemented as a front-end to the *DLV* logic programming system.

[19] presents a language about actions using causal laws to reason in probabilistic settings and solves the planning problems in such settings. The language resembles similarities to those described above, but the action theory incorporates probabilities and probabilistic reasoning techniques to solve the planning problems with uncertainty.

Dimopoulos, Nebel and Köhler ([2]) were the first to present a framework for encoding planning problems in logic programs with negation-as-failure and implementing it using an ASP engine. In this work, the idea is the same as ours, that is, the models of the logic program correspond to the plans. However, their work considers action-based planning problems and incorporates ideas from such planners *GRAPHPLAN* and *SATPLAN*. In terms of the underlying assumptions and methods presented in [2], our approach is completely different. Both methods complement each other.

[20] discusses solving planning programs by logic programs. The difference between this work and the one described above is that [20] incorporates domain-dependent control knowledge to improve the performance of the planning. In this respect, this work is similar to HTN planning algorithms. However, the encoding provided in this work is conceptually not an HTN planner.

2 Definitions for HTN Planning: Syntax and Semantics

A *term* is either a constant or a variable symbol. A *state* $\mathcal{S}$ is a set of ground atoms. An *axiom* is an expression of the form $a \leftarrow l_1, \ldots, l_n$, where a is an atom and the l_i are literals. Axioms need not be ground. We assume that the set of axioms does not contain cycles through negation.

A task is an expression of the form $(ht_1t_2 \ldots t_n)$, where h (the task's name) is a task symbol, and $t_1, t_2, \ldots, t_n$ (the task's arguments) are terms. A task can be either primitive or composite. A task list is a list of tasks.

An *operator* is an expression of the form $\mathsf{Op} = \mathbf{(Op}\ h\ \chi_{del}\ \chi_{add}\mathbf{)}$, where h (the *head*) is a primitive task and χ_{add} and χ_{del} are lists of atoms (called the *add-* and *delete-lists*). The set of variables in the atoms in χ_{add} and χ_{del} is a subset of the set of variables in h. Let t be a primitive task, $\mathsf{Op} = \mathbf{(Op}\ h\ \chi_{del}\ \chi_{add}\mathbf{)}$ be an operator, and $\mathcal{S}$ be the current state of the world. Suppose that u is a unifier for h and t. Then the operator instance Op^u is *applicable* to t in $\mathcal{S}$ and the result of applying it to t in $\mathcal{S}$ is a new state result$(\mathsf{Op}^u, \mathcal{S})$ that is created by first deleting every ground atom in χ^u_{del} from $\mathcal{S}$ and then by adding every ground atom in χ^u_{add} to $\mathcal{S}$.

A *method* is an expression of the form $\mathbf{(Meth}\ h\ \chi\ \mathbf{t)}$ where h (the method's *head*) is a compound task, χ (the method's *preconditions*) is a conjunct and $\mathbf{t}$ is

a totally ordered list of subtasks, called the *task list*. Let t be a compound task, $\mathcal{S}$ be the initial state, $Meth = \mathbf{(Meth}\ h\ \chi\ \mathbf{t)}$ be a method, and $\mathcal{AX}$ be an axiom set. Suppose that u is a unifier for h and t, and that v is a unifier that unifies χ^u with respect to $\mathcal{S} \cup \mathcal{AX}$. Then the method instance $(Meth^u)^v$ is *applicable* to t in $\mathcal{S}$, and the result of applying it to t is the task list $\mathbf{r} = (\mathbf{t}^u)^v$. The task list $\mathbf{r}$ is a *simple reduction* of $\mathbf{t}$ by $Meth$ in $\mathcal{S}$.

A *plan* is a list of heads of ground operator instances. If $P = (p_1 p_2 \dots p_n)$ is a plan and $\mathcal{S}$ is a state (a set of ground atoms a), then the *result* of applying P to $\mathcal{S}$ is the state $\text{result}(\mathcal{S}, P) = \text{result}(\text{result}(\dots(\text{result}(\mathcal{S}, p_1), p_2), \dots), p_n)$. A plan P is called a *simple plan* when $n = 1$.

A *planning domain* is a set of axioms, operators and methods. A planning domain can contain more than one method applicable to a particular compound task, but it must have only one operator applicable to a particular primitive task. A *planning problem* is a triple $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is a state, $\mathbf{t}= (t_1 t_2 \dots t_k)$ is a task list, and $\mathcal{D}$ is a planning domain.

Suppose $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is a planning problem and $P = (p_1 p_2 \dots p_n)$ is a plan. We say that P solves $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, or equivalently, that P *achieves* $\mathbf{t}$ from $\mathcal{S}$ in $\mathcal{D}$ (we will omit the phrase "in $\mathcal{D}$" if the identity of $\mathcal{D}$ is obvious) if any of the following cases is true: (1) $\mathbf{t}$ and P are both empty, (i.e., $k = 0$ and $n = 0$); (2) t_1 is a primitive task, p_1 is a simple plan for t_1, $(p_2 \dots p_n)$ achieves $(t_2 \dots t_k)$ from $\text{result}(\mathcal{S}, p_1)$; and (3) t_1 is a composite task, and there is a simple reduction $(r_1 \dots r_j)$ of t_1 in $\mathcal{S}$ such that P achieves $(r_1 \dots r_j t_2 \dots t_k)$ from $\mathcal{S}$. The planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is *solvable* if there is a plan that solves it. We define $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ as the set of all possible plans that can be found given $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ as a solution during planning. Note that $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is a *multi set*: it can contain the same plan in a number of copies. This is because the same plans may be generated during the planning process due to the fact that there may be different method applicable to a particular compound task, thus creating a branching point in the search space of the planner, and different branches may end up with same plans.

3 Encoding HTN Planning in Nonmonotonic LP

Our approach of encoding HTN-planning problems as logic programs is based on *SHOP*'s representation of a planning problem.[3] In this section, we present first steps of a causal theory of HTN planning based on the *SHOP* formalism. The reason for presenting this causal theory is not to give a formal semantics, but to give some motivations for the more technical aspects of the translation methodology presented in the later in this section.

Definition 1 (Causable Tasks). *For a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the notion of an ordered list of tasks to be causable* wrt. *$(\mathcal{S}, \mathcal{D})$ comes in 3 steps.*

[3] *SHOP* is a domain-independent HTN-planning system that plans for tasks in the same order that they will later be executed. This provides the planner with a significant inferencing and reasoning power, including the ability to call external programs and the ability to perform numeric computations. Due to the lack space, we cannot go into the details of *SHOP* in this paper; for more information please see [11, 21].

Literals: *A literal l is caused by $(\mathcal{S}, \mathcal{AX})$ if l is true in all answer sets of $\mathcal{S} \cup \mathcal{AX}$, where $\mathcal{AX}$ is the set of axioms in $\mathcal{D}$. A literal l is causable wrt. $(\mathcal{S}, \mathcal{D})$ if it is caused by $(\mathcal{S}, \mathcal{AX})$ A conjunction of literals is causable wrt. $(\mathcal{S}, \mathcal{D})$ if every conjunct is causable wrt. $(\mathcal{S}, \mathcal{D})$.*

Primitive tasks: *An ordered list of primitive tasks $t_1, \ldots, t_n$ is causable wrt. $(\mathcal{S}, \mathcal{D})$ if the following holds:*

*For each t_i, there exists an operator (**Op** h χ_{del} χ_{add}) $\in \mathcal{D}$ such that there is a unifier u for h and t_i.*

This includes that the empty list [] is causable.

Composite tasks: *An ordered list of tasks $t_1, \ldots, t_j, \ldots, t_n$, where t_j is a composite task and all tasks $t_1, \ldots, t_{j-1}$ are primitive tasks, is causable wrt. $(\mathcal{S}, \mathcal{D})$ if the following holds:*

1. *there exists a method (**Meth** h χ $\{t_{j_1}, \ldots, t_{j_m}\}$) $\in \mathcal{D}$ for t_j such that there is a unifier u for h and t_j,*
2. *the preconditions-list χ^u, which is a list of literals representing a conjunction, is causable wrt. $(result(\mathcal{S}, t_1, \ldots t_{j-1}), \mathcal{D})$, and*
3. *the ordered list $(t_1, \ldots, t_{j-1}, t_{j_1}, \ldots, t_{j_m}, t_{j+1}, \ldots t_n$ is causable wrt. $(\mathcal{S}, \mathcal{D})$.*

Theorem 1. *Let a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be given, where $\mathcal{S}$ is the initial state, $\mathbf{t}$ is the list of tasks to be achieved and $\mathcal{D}$ is the domain description.*

There is a solution to $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ if and only if *the list $\mathbf{t}$ is causable wrt. $(\mathcal{S}, \mathcal{D})$.*

Using this causal theory as an intermediate step, we developed a systematic translation method for mapping planning problems to logic programs with negation which we illustrate now.

Translating a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ to its logic program counterpart $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ requires encoding the methods, the operators, and the axioms as logic program segments as well as the underlying ordered task decomposition characteristics of *SHOP*.

Definition 2 ($\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$: Translation for the Planning Problem). *The logic program $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ that solves the planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is defined as*

$$\begin{aligned}\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D})) = \mathfrak{Trans}(G) \ \cup \ \mathfrak{Trans}(\mathcal{S}) \ \cup \ \mathfrak{Trans}(\mathbf{t}) \ \cup \ \mathfrak{Trans}(\mathcal{AX}) \cup \\ \mathfrak{Trans}(F) \ \cup \ \mathfrak{Trans}(\mathcal{OP}) \ \cup \ \mathfrak{Trans}(\mathcal{METH}), \cup \mathfrak{Trans}(ST),\end{aligned}$$

where each $\mathfrak{Trans}(\ldots)$ is a logic program segment as defined in the following subsections.

Encoding the Grounding Rules Given a planning problem, these rules encode all of the objects that may be used in solving the planning problem, the type descriptions of those objects, and all of the atoms that may appear to be true in some state of the planner during the planning process. The reason that we need these rules are the following important distinction between *SHOP* and most nonmonotonic systems: *SHOP* allows using variables in the domain descriptions

of the planning problems and all variables are implicitly universally quantified. However, unlike most nonmonotonic systems, *SHOP* searches over the original formulas without expanding the ground representation before search.

Definition 3 ($\mathfrak{Trans}(G)$: Translation for the Grounding Rules).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, we define $\mathfrak{Trans}(G)$ as the logic program segment that consists of the following set of rules:

- *For each object o:* $[type](o) : -$
- *For each atom A:* $atom(A) : -$

Encoding the Initial State. The initial state $\mathcal{S}$ is a set of ground atoms.

Definition 4 ($\mathfrak{Trans}(\mathcal{S})$: Translation for Initial State).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for each ground atom $a \in \mathcal{S}$, the logic program segment $\mathfrak{Trans}(\mathcal{S})$ contains the rule "$in\_state(a, 0) : -$", where 0 indicates the initial time.

Encoding the Goal Task(s). In *SHOP*-like HTN planning, a task is accomplished if and only if it is *causable* with respect to the initial state and the domain description given in the planning problem. This is due to the Definition 1 and a direct consequence of Theorem 1. We denote the fact that whether a task is causable by the following definition.

Definition 5 (CAUSABLE).
Given a task t, we define $CAUSABLE(t, T_{selected}, T_{accomplished})$ as follows:

$$\begin{cases} false & \text{if } t \text{ is a primitive task and} \\ & \text{there is no operator for it in } \mathcal{D}, \text{ or} \\ & \text{if } t \text{ is a compound task and} \\ & \text{there is no method for it in } \mathcal{D}, \\ currentTask(t, T_{selected}), & \text{if } t_k \text{ is a primitive task and} \\ & \text{there is an operator for it in } \mathcal{D}, \\ causable(t, T_{selected}, T_{accomplished}) & \text{if } t_k \text{ is a compound task and} \\ & \text{there is a method for it in } \mathcal{D}, \end{cases}$$

where the predicate $currentTask(t, T)$ encodes the fact that the task t is selected as the "current task" – i.e., the task that the planner will try to accomplish next – at time T.

We are now ready to define the logic program that encodes the goal task list of a given planning problem.

Definition 6 ($\mathfrak{Trans}(\mathbf{t})$: Translation for Goal Tasks).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, let $\mathbf{t} = h_1, h_2, \ldots, h_n$ be the ordered sequence of goal tasks. Then, $\mathfrak{Trans}(\mathbf{t})$ is the logic program segment that contains one rule for each goal task h_i, where $i = 1, 2, \ldots, n$, as follows:

$$\begin{aligned} &currentTask(h_1, 0) \ : - \\ &currentTask(h_i, T_i) : - \ CAUSABLE(h_{i-1}, T_{i-1}, T_i), T_i > T_{i-1}. \end{aligned}$$

Note that if there exists only one goal task to be accomplished for the problem in hand, then only defining the first rule will suffice. Definition 6 enforces the fact that a goal task h_i is designated as the current task to be accomplished if the previous goal task h_{i-1} in $\mathbf{t}$ is causable. This is a direct consequence of our Theorem 1. The planning process terminates successfully when all of the goal tasks are accomplished (i.e., caused) in the order they are given in the planning problems. The following definition is given to encode the successful termination of the planning process.

Definition 7 ($\mathfrak{Trans}(ST)$: Successful Termination).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the logic program segment $\mathfrak{Trans}(ST)$ that encodes the successful termination of the planning process (i.e., the fact that a solution to the given planning problem is found) is defined as follows:

$$\begin{array}{l} plan\_found :- CAUSABLE(h_n, T_n, T_{n+1}). \\ \qquad\qquad\quad :- \ not\, plan\_found. \end{array}$$

Encoding the Domain Control Structures. Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the domain description $\mathcal{D}$ contains axioms, operators and methods as described in the previous section. For each of these constructs, we present a translation procedure.

Definition 8 (Translation for Literals).
Given a literal, l, we define $C(l, T)$, the translation of l at time T (a is an atom):

$$C(l,T) := \begin{cases} in\_state(a,T) & \text{if } l = a, \\ not\ in\_state(a,T) & \text{if } l = \neg a. \end{cases}$$

Definition 9 ($\mathfrak{Trans}(\mathcal{AX})$: Translation for Axioms).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for all "$a \leftarrow l_1, \ldots, l_n$" $\in \mathcal{AX}$, the logic program segment $\mathfrak{Trans}(\mathcal{AX})$ contains the following rule

$$in\_state(a,T) \ :- C(l_1,T), C(l_2,T), \ldots, C(l_n,T),$$

where $C(l_i, T)$ is the translation of a literal as defined in Definition 8 above.

Definition 10 ($\mathfrak{Trans}(\mathcal{OP})$: Translation for Operators).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for all $\mathsf{Op} \in \mathcal{OP}$, $\mathfrak{Trans}(\mathsf{Op})$ is the logic program segment that contains the following rules:

for all $a \in Del(\mathsf{Op})$: $out\_state(a, T+1) \ :- currentTask(h, T)$.
and for all $a \in Add(\mathsf{Op})$: $in\_state(a, T+1) \ :- currentTask(h, T)$.

Note that an operator only describes the *change* it causes to occur in the current state. Therefore, we still need to address the famous *Frame Problem* as follows.

Definition 11 ($\mathfrak{Trans}(F)$: Keeping Track of the State $\mathcal{S}$). *The logic program segment $\mathfrak{Trans}(F)$ that encodes the frame axiom is defined as follows:*

$$in\_state(A, T+1) :- atom(A), in\_state(A, T), not\, out\_state(A, T+1).$$

Definition 12 ($\mathfrak{Trans}(\mathcal{METH})$: Translation for Methods). *Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, let h be a compound task that needs to be accomplished in the solution of the given planning problem. Suppose the domain description $\mathcal{D}$ contains N methods whose heads unify with h; namely, m_1, m_2, ..., m_N. Let $\mathtt{Pre}(h)_i$ be the label for the precondition list of the method m_i. Then, the logic program segment that encodes these methods is defined as follows:*

1. The nondeterministic choice of which method to apply to the task h:

$$\begin{aligned}
method_1(h, \mathtt{Pre}(h)_1, T) \;:-\; & currentTask(h, T), \\
& not\, method_2(h, \mathtt{Pre}(h)_2, T), ..., \\
& not\, method_N(h, \mathtt{Pre}(h)_N, T) \\
\vdots \qquad\qquad \vdots \qquad & \qquad\qquad \vdots \\
method_N(h, \mathtt{Pre}(h)_N, T) \;:-\; & currentTask(h, T), \\
& not\, method_1(h, \mathtt{Pre}(h)_1, T), ..., \\
& not\, method_{N-1}(h, \mathtt{Pre}(h)_{N-1}, T)
\end{aligned}$$

2. The precondition list χ_i of each method m_i: *For each precondition $p \in \chi_i$, we have one of the following two cases:*
   (a) p is a positive literal and it contains free variables: *The free variables in a precondition literal are the variable symbols that do not appear in the head of the method m_i. We denote p as $p = p(Y_1, Y_2, \ldots, Y_f)$, where $Y_1, Y_2, \ldots, Y_f$ are the free variables in p.*[4] *Let R_j denote the range of the free variable Y_j – i.e. the set of all possible values for the variable Y_j –, and for each such variable Y_j, let $Y_{j,k}$ be a new variable symbol such that $k = 1, \ldots, R_j$. Then, $\mathfrak{Trans}(\mathcal{METH})$ contains the following rule to encode the precondition $p \in \chi_i$:*

$$\begin{aligned}
checked_state(p(Y_{1,1}, Y_{2,1}, \ldots, Y_{f,1}), T) \;:-\; & \\
& method_i(h, \mathtt{Pre}(h)_i, T), \\
& in_state(p(Y_{1,1}, Y_{2,1}, \ldots, Y_{f,1}), T), \\
& not\, checked_state(p(Y_{1,1}, Y_{2,1}, \ldots, Y_{f,2}), T), \\
& not\, checked_state(p(Y_{1,1}, Y_{2,1}, \ldots, Y_{f,3}), T), \\
& \quad \vdots \\
& not\, checked_state(p(Y_{1,R_1}, Y_{2,R_2}, \ldots, Y_{f,R_f}), T), \\
& \textstyle\bigwedge_{j=1}^{f} Y_{j,1}! = Y_{j,2}! = \ldots ! = Y_{j,R_j}.
\end{aligned}$$

[4] Note that p may also contain variable symbols that do appear in the head of the particular method. However, those variables are not relevant for the discussion above, so we omitted them for the sake of simplicity. Normally, those variables appear in the translated logic programs.

(b) Otherwise: *The logic program segment* $\mathfrak{Trans}(\mathcal{METH})$ *contains the rule "* $checked\_state(p,T) \ :- \ C(p,T), method_i(h, \mathtt{Pre}(h)_i, T)$*" (where* $C(p,T)$ *is as defined in Definition 8) to encode the precondition* $p \in \chi_i$.

3. The decomposition list $\{t_1, t_2, \ldots, t_n\}$ for m_i: *Let* $p_1, p_2, \ldots, p_{|\chi_i|}$ *be the list of preconditions of the method* m_i. *Then, the logic program segment* $\mathfrak{Trans}(\mathcal{METH})$ *contains the following rules to encode the decomposition list of* m_i *(note that the time variable* T_1 *in the following rule definitions in this item denote the same value as the time variable* T *in the rule definitions presented in other items does):*

$$
\begin{array}{lll}
currentTask(t_1, T_1) & :- & method_i(h, \mathtt{Pre}(h)_i, T_1), \\
 & & \bigwedge_{k=1}^{|\chi_i|} checked\_state(p_k, T_1). \\
currentTask(t_2, T_2) & :- & method_i(h, \mathtt{Pre}(h)_i, T_1), \\
 & & \bigwedge_{k=1}^{|\chi_i|} checked\_state(p_k, T_1), \\
 & & CAUSABLE(t_1, T_1, T_2), \\
 & & T_2 > T_1. \\
\vdots & \vdots & \vdots \\
currentTask(t_n, T_n) & :- & method_i(h, \mathtt{Pre}(h)_i, T_1) \\
 & & \bigwedge_{k=1}^{|\chi_i|} checked\_state(p_k, T_1), \\
 & & CAUSABLE(t_{n-1}, T_{n-1}, T_n), \\
 & & T_n > T_{n-1}.
\end{array}
$$

4. The accomplishment (i.e., causation) of h by the method m_i:

$$
\begin{array}{lll}
causable(h, T_1, T_{n+1}) & :- & method_i(h, T_1), \\
 & & \bigwedge_{k=1}^{|\chi_i|} checked\_state(p_k, T_1), \\
 & & CAUSABLE(t_n, T_n, T_{n+1}), \\
 & & T_{n+1} > T_n.
\end{array}
$$

4 Results: Theory and Practice

In this section, we present our theoretical results on the correctness of our translation method and the soundness and the completeness of the resulting logic programs as planning systems as well as the experiments we have undertaken. Due to space limitations, we will not present the whole proofs here, the basic ideas behind them can be found in [22].

4.1 Soundness and Completeness

Given an HTN-planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ for *SHOP*, let $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ be the translated logic problem with negation as described in the previous section. We are interested in the relationship between the models (or answer sets) of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ and the solutions to the planning problem.

Soundness and completeness are the two important requirements for any planning system. Soundness means that all of the plans that are generated by

the planner are actually true solutions to the given planning problem; that is, no plan, which is not solution to the problem, should be generated. Completeness means that the planning system must be able to generate all of the possible plans (solutions) for the given problem.

Theorem 2 (Soundness and Completeness of ASP using $\mathfrak{Trans}(\cdot)$).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is the initial state, $\mathbf{t}$ is the list of tasks to be achieved and $\mathcal{D}$ is the domain description, let $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ be the corresponding logic program with negation. Furthermore, let $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be the set of solutions of the planning problem.

Then, the answer sets of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ correspond exactly to the plans in $\mathbf{Sol}$ $(\mathcal{S},\mathbf{t},\mathcal{D})$. There is a bijection between these two sets and each plan in $\mathbf{Sol}$ $(\mathcal{S},\mathbf{t},\mathcal{D})$ can be reconstructed from its corresponding answer set in $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ and vice versa.

4.2 Experimental Study

In our experiments, we used three different planning domains:

The Travelling Domain: This domain is the one of the domains included in the distribution of SHOP planning system. The scenario for the domain is that we want to travel from one location to another in a city. There are three locations: downtown, uptown, and park. There are three possible means of transportation: taxi, bus and foot. The planning problem is to generate a sequence of actions that needs to be taken in a trip from our original location to our destination by using the available transportation means. More detailed description of this domain is given in [22]

The Miconic-10 Elevator Domain: This domain was introduced as an official benchmark domain during the AIPS-2000 competition (see [23] and `http://www.cs.toronto.edu/aips2000`). Its simplest version (the one referred to as the "first track" version at `http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html`) was one of the test cases in [20], and we used the same version in our experiments. In this version, the planner simply has to generate plans to serve a group of passengers of whom the origin and destination floors are given. There are no constraints such as satisfying space requirements of passengers or achieving optimal elevator controls.

The Zeno-Travel Domain: The Zeno-Travel problem was one of the domains that were introduced as recent benchmarks in International Planning Competition (IPC-2002).[5] This domain is again a transportation domain that-involves transporting people from their original locations to their destinations via planes using two different modes of movement: namely fast and slow. There were four versions of the domain in the competition. In our

[5] IPC-2002 was organised within the Sixth International Conference on AI Planning and Scheduling 2002 (AIPS-2002).

experiments we used the simplest version. For more information on IPC-2002, please see <http://www.dur.ac.uk/d.p.long/competition.html>. For more information about the ZenoTravel planning problem, please see <http://www.cs.washington.edu/ai/zeno.html>.

We prepared two sets of experiments: the first set aimed for investigating the time performance of the logic programs generated by our translation methodology and the second set was for investigating the effects of grounding on their performance. We ran our experiments on an HP OmniBook 6000 Laptop with 128MB RAM and an Intel Pentium III 600 Mhz processor. We used both the software package *smodels* v2.7—which is available at `http://www.tcs.hut.fi/Software/smodels/`—and the *DLV* system—which is available at `http://www.dbai.tuwien.ac.at/proj/dlv/`—as testing environments for our logic program encodings.

Efficiency of Encoding HTN Control Knowledge In this set of experiments, we compared the time performance of the logic programs produced by using our translation methodology with that of the logic-program encodings presented in [20]. In their paper, Son et al., showed that encoding control knowledge has increased the time performance of the logic programs for solving planning problems. The encoding methods proposed in [20], however, does not use actual HTN control knowledge, rather they make use of only a few properties of HTNs —as they are introduced in [9]— for implementing control knowledge in logic programs that perform action-based planning.

In our experiments, we aimed to investigate the impact of using HTN control knowledge as used in *SHOP* on the performance of logic programs that perform planning, and we compared our results with those of [20]. The problems that we used in these experiments are from `http://www.CS.NMSU.Edu/~tson/asp_planner`. Table 1 shows both our results and the results from [20], which were obtained on the *smodels* system.

Table 1. Comparison of HTN Encoding, $\mathfrak{Trans}(\cdot)$, on *smodels* and *DLV* with on Miconic-10 problems. All times are in CPU seconds.

Problem	***smodels***	***DLV***	**[20]**
S1-0	0.050	0.040	0.520
S2-0	0.330	0.060	12.410
S3-0	1.390	0.080	121.810
S4-0	4.540	0.260	883.700
S5-0s1	19.530	0.640	no solution
S5-0s2	20.630	0.680	no solution
S6-0	23.150	0.980	no solution

Table 2. Comparison of HTN Encoding, $\mathfrak{Trans}(\cdot)$, on *smodels* and *DLV* with *SHOP* on Travelling problems. All times are in CPU seconds.

Problem	***smodels***	***DLV***	***SHOP***
P1	3.23	0.20	0.026
P2	2.23	0.12	0.002
P3	2.19	0.22	0.003
P4	2.08	0.10	0.002
P5	2.20	0.19	0.004
P6	2.18	0.11	0.009
P7	2.21	0.19	0.003
P8	2.15	0.08	0.003

Table 3. Comparison of *DLV* with *SHOP* on ZenoTravel Domain. All times are in CPU seconds.

Problem	*smodels*	*DLV*	*SHOP*	Performance Ratio(*DLV* / *SHOP*)
P2	no-solution	0.670	0.010	67.00
P4	no-solution	0.320	0.010	32.00
P8	no-solution	26.180	0.030	872.67
P9	no-solution	38.390	0.070	548.43
P12	no-solution	22.930	0.020	1146.50
P13	no-solution	16.560	0.060	276.00
P16	no-solution	78.060	0.090	867.34
P19	no-solution	146.030	0.120	1216.92
P20	no-solution	168.630	0.130	1297.15
P23	no-solution	4275.25	12.250	349.00
P24	no-solution	3612.96	7.980	452.75

The results clearly show that the logic programs produced by our translation methodology outperform the logic programs produced in [20]. Our encoding was even able to solve a problem, for a solution could not be found by [20]. In this respect, these results that *SHOP*-like HTN planning is an effective way of solving planning problems.

The Effects of Grounding We hypothesise that our translation methodology provides more efficient logic programs with ASP semantics if the system on which those programs are implemented allows the usage of variables in the programs– that is, if the ASP systems do not require solely ground programs as input, but can work with variables in the programs and ground those variables as the search progresses. Most of the recent planning systems—such as *SHOP* [11], *TALPlanner* [24], etc.—can work on planning-problem descriptions with variables and these systems are proven to be faster than those which require ground descriptions.

As we described earlier, the *smodels* system cannot work on the logic programs with variables. To test our hypothesis, we applied our translation methodology to our elevator and travelling examples to produce logic programs on a different system called *DLV*. *DLV* is a deductive database system, and can be used as a logic programming system as well. It implements stable model semantics and it supports the usage of variables in the input logic programs to some extent.

Table 2 show our results on the problems of the Simple-Travel domain. As it can be seen, our programs are much more faster on *DLV*, than on *smodels*. Like *smodels*, *DLV* also imposes a safeness restriction on the input programs, but since it allows the usage of variables in the input programs, we do not have to specify the ranges of the variables as long as they do not violate the safeness restrictions required by the system, which is an important characteristic of *DLV*'s approach to grounding.

On the elevator problems, however, the performances of our programs are almost the same (see Table 1). This is because the encodings for these problems are mostly ground; they did not require using variables. Thus, we were not able to observe the effect of grounding techniques used by the two systems on the performance of our programs for these problems.

On the problems from the Zeno-Travel domain, *smodels* was not able to solve any of the problems because of memory limitations. Table 3 shows our results on *DLV* as well as on *SHOP*. In these experiments, we investigated the ratio between the amount of time that our logic programs require and the time required by *SHOP*. If the average-case time complexity of our programs were worse than that of *SHOP*, then we would expect this ratio would get worse with increasing problem size. However, it did not seem to be the case, as it can be seen in Table 3. Although there is not enough data to say so conclusively, our results suggest that the average-case time complexity of our programs may be roughly the same as that of *SHOP*. This gives reason to hope that future improvements in our programs and in ASP solvers may make it possible to get performance competitive with planning systems such as *SHOP*.

5 Conclusions and Future Research Directions

In this paper, we have described a way to encode HTN-planning problems as logic programs under the answer set semantics. This transformation is sound and complete, and it corresponds to HTN-planning systems that generate plans by using ordered task decomposition. In the view of the latter, our method differs from the previous approachesfor encoding planning problems as logic programs (as first introduced in [2] and further investigated in [20] by encoding control knowledge to increase the performance of the logic programs).

Our overall aim was to investigate to what extent state-of-the-art nonmonotonic theorem provers can compete with dedicated planners (in particular those based on HTN) and what lessons we could learn from the different translation methods. In our experiments, we used our approach to create both *smodels* and *DLV* logic programs on three different AI planning domains: the Simple-Travel Domain, the "first track" version of the Miconic 10 Elevator Planning Domain, and the simplest version of the ZenoTravel Domain. Here is a summary of our experimental results and what we believe they signify:

1. Although the experiments we have done so far were on relatively simple planning domains, the results were encouraging since they showed the possibility that encoding the HTN control knowledge in ASP programs may provide efficient solutions to planning problems. In the near future, we are planning to conduct further experiments on more complicated HTN planning domains to test this hypothesis.
2. Our experiments suggests that the average-case time complexity of our programs may be roughly the same as that of *SHOP*. Although we do not have enough data to say so conclusively, this gives reason to hope that future

improvements in our programs and in ASP solvers may make it possible to get performance competitive with planning systems such as *SHOP*.

3. Our experiments showed that the way that ASP systems perform grounding is an important factor on the performance of the logic programs. In our experiments, our ASP programs were slower than *SHOP*. Our explanation for this difference in performance is the following: *SHOP*, like most planners, can work on planning-problem descriptions with variables. However, the ASP systems we used requires safeness constraints and they are creating ground instances of clauses that are irrelevant for the planning process in order to meet these constraints.

As a byproduct, like [2], we believe our method can be easily used as transferring benchmarks from the planning community to benchmarks for comparing nonmonotonic systems based on computing answer sets.Furthermore, our method complements the technique described in [2] as it enables to transfer benchmarks for HTN-planning problems. In the near future, we will conduct more experiments on translating more complicated HTN domains than the ones presented in this paper.

Acknowledgments

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, and Naval Research Laboratory N00173021G005. The opinions expressed in this paper are those of authors and do not necessarily reflect the opinions of the funders.

References

1. Niemelä, I., Simons, P.: Efficient Implementation of the Well-founded and Stable Model Semantics. In Maher, M., ed.: Proceedings of the Joint International Conference and Symposium on Logic Programming, Bonn, Germany, The MIT Press (1996) 289–303
2. Dimopoulos, Y., Nebel, B., Koehler, J.: Encoding Planning Problems in Nonmonotonic Logic Programs. In Steel, S., Alami, R., eds.: Proceedings of the Fourth European Conference on Planning, Toulouse, France, Springer-Verlag (1997) 169–181
3. Dix, J., Furbach, U., Niemelä, I.: Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In Voronkov, A., Robinson, A., eds.: Handbook of Automated Reasoning. Elsevier-Science-Press (2001) 1121–1234
4. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Cohn, A.G., Schubert, L., Shapiro, S.C., eds.: Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann (1998) 406–417
5. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74

6. Rao, P., Sagonas, K., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In Dix, J., Furbach, U., Nerode, A., eds.: Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference. LNAI 1265, Berlin, Springer (1997) 430–440
7. Apt, K.R., Marek, V., Truszczynski, M., Warren, D.S., eds.: The Logic Programming Paradigm: Current Trends and Future Directions, Berlin, Springer (1999)
8. Sacerdoti, E.: A Structure for Plans and Behavior. American Elsevier Publishing (1977)
9. Erol, K., Hendler, J., Nau, D.: UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In Hammond, K., ed.: Proceedings of AIPS-94, Chicago, IL, AAAI Press (1994) 249–254
10. Wilkins, D.: Practical Planning - Extending the Classical AI Planning Paradigm. Morgan Kaufmann (1988)
11. Nau, D., Cao, Y., Lotem, A., Muñoz-Avila, H.: SHOP: Simple Hierarchical Ordered Planner. In Dean, T., ed.: Proceedings of IJCAI-99, Morgan Kaufmann (1999)
12. Nau, D., Cao, Y., Lotem, A., Muñoz-Avila, H., Mitchell, S.: Total-Order Planning with Partially Ordered Subtasks. In Nebel, B., ed.: Proceedings of IJCAI-01, Seattle, Washington, Morgan Kaufmann (2001) 425–430
13. Gelfond, M., Lifschitz, V.: Action Languages. Electronic Transactions on AI **2** (1998) 193–210
14. Turner, H.: Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. The Journal of Logic Programming **31** (1997) 245–298
15. Lifschitz, V.: Action Languages, Answer Sets and Planning. In Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S., eds.: The Logic Programming Paradigm: A 25-Year Perspective, Springer-Verlag (1999) 357–373
16. McCain, N., Turner, H.: Causal Theories of Action and Change. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97), Menlo Park, CA, AAAI Press (1997) 460–465
17. McCain, N.: Using Causal Calculator with the C Input Language. Technical report, University of Texas at Austin (1999)
18. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: The DLV-K Planning System: Progress Report. In Ianni, G., Flesca, S., eds.: Proceedings of Journees Europeens de la Logique en Intelligence Artificielle (JELIA '02). LNCS 2424, Springer (2002) 541–544
19. Baral, C., Tran, N., Tuan., L.: Reasoning about Actions in a Probabilistic Setting. In: AAAI/IAAI 2002, AAAI Press (2002) 507–512
20. Son, T., Baral, C., McIlraith., S.: Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In Eiter, T., Truszczyński, M., Faber, W., eds.: Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference. LNCS 2173, Berlin, Springer (2001) 226–239
21. Nau, D., Au, T.C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., Yaman, F.: SHOP2: an HTN planning system. Journal of Artificial Intelligence Research (2003) To appear.
22. Dix, J., Kuter, U., Nau, D.: HTN Planning in Answer Set Programming. Technical Report CS TR 4336, University of Maryland (2002) Submitted for publication.
23. Bacchus, F.: AIPS'00 Planning Competition. AI Magazine **22** (2001) 47–56
24. Kvarnström, J., Doherty, P.: TALplanner: A Temporal Logic Based Forward Chaining Planner. Annals of Mathematics and Articial Intelligence **30** (2001) 119–169