# Planning in Answer Set Programming using Ordered Task Decomposition

JÜRGEN DIX AND UGUR KUTER AND DANA NAU [1]

ABSTRACT. In this paper we investigate a formalism for solving planning problems based on *ordered task decomposition* using *Answer Set Programming* (ASP). Our planning methodology is an adaptation of *Hierarchical Task Network* (HTN) planning, an approach that has led to some very efficient planners. The ASP paradigm evolved out of the stable semantics for logic programs in recent years and is strongly related to nonmonotonic logics. It also led to various very efficient implementations (*Smodels*, *DLV*). While all previous approaches for using ASP for planning rely on *action-based* planning, we consider for the first time a formulation of HTN planning as described in the *SHOP* planning system and define a systematic translation method from *SHOP*'s representation of a planning problem into a logic program 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. Our approach does not rely on a particular system for computing answer sets and serves several purposes. (1) It constitutes a means to evaluate ASP systems by using well-established benchmarks from the planning community. (2) It makes the more expressive HTN planning available in ASP. (3) When our approach is implemented on ASP solvers, its time requirement appears to grow no faster than roughly proportional to that of a dedicated HTN planning system (*SHOP*). (4) It outperforms the transformation of an STRIPS-style planning problem into ASP proposed in [Son *et al.*, 2001]. The particular relevance of that transformation method to our work is that, in their work, [Son *et al.*, 2001] proposed to use a form of control knowledge to speed up the classical planning process. In this paper, we show that HTN control knowledge provides more time-efficient transformations compared to the control strategies presented in [Son *et al.*, 2001].

[1] Authors' addresses:
Jürgen Dix, Technical University of Clausthal, Institut für Informatik, Julius-Albert-Str. 4, D–38678 Clausthal, Germany.
Ugur Kuter and Dana Nau, University of Maryland, Dept. of CS, College Park, MD 20752, USA.

## 1 Introduction

In the past few years, the availability of 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 systems was *Smodels* [Niemelä and Simons, 1996] and one of the early applications [Dimopoulos *et al.*, 1997] was to transform planning problems in a suitable way and to run *Smodels* on them (see also [Dix *et al.*, 2001]).

Since then, additional systems with different properties for dealing with logic programs have become available: *DLV* [Eiter *et al.*, 1998], *XSB* [Chen and Warren, 1996; Rao *et al.*, 1997], to cite the most well-known. In addition, the paradigm of Answer Set Programming (ASP) has emerged (put forth in [Niemelä, 1999; Marek and Truszczyński, 1999], see also [Apt *et al.*, 1999]). It is based on two key ideas: (1) to solve 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) to tackle the problems located on the second level of the polynomial hierarchy, which seem well suited for the machinery of answer sets. In particular, many planning problems fit in this picture.

In this paper, we investigate how to formulate and solve HTN planning problems using nonmonotonic logic programs under the ASP semantics. HTN planning is an AI-planning paradigm in which the goals of the planner are defined in terms of activities (tasks) and the planning process is performed by using the techniques of task decomposition.

HTN planning was first proposed more than 25 years ago [Sacerdoti, 1990; Tate, 1977]. Historically, most of the HTN planning research has focused on specific application domains. Examples include production-line scheduling [Wilkins, 1988], crisis management and logistics [Currie and Tate, 1991; Tate *et al.*, 1994; Biundo and Schattenberg, 2001], planning and scheduling for spacecraft [Aarup *et al.*, 1994; Estlin *et al.*, 1997], equipment configuration [Agosta, 1995], manufacturability analysis [Hebbar *et al.*, 1996; Smith *et al.*, 1997], evacuation planning [Muñoz-Avila *et al.*, 2001], and the game of bridge [Smith *et al.*, 1998a; 1998b]. However, there are several domain-independent HTN planning systems, such as *Nonlin* [Tate, 1977], *Sipe-2* [Wilkins, 1990], *O-Plan* [Currie and Tate, 1991; Tate *et al.*, 1994], *UMCP* [Erol *et al.*, 1994], *SHOP* [Nau *et al.*, 1999], *ASHOP* [Dix *et al.*, 2003; 2002], and *SHOP2* [Nau *et al.*, 2001].

In this work, we focus on the *SHOP* planning system, which is a domain-independent HTN planning system that is built around a concept called *ordered task decomposition*. In particular:

- 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 captures the solutions (plans) of the planning problem.
- We establish soundness and completeness results for our method: answer sets of the transformation are in one-to-one correspondence with solutions of the original planning problem.
- We propose to use established benchmarks for planning problems as benchmarks for testing ASP systems, by transforming the former using our translation into logic programs.
- Although we describe our transformation using the syntax of the *Smodels* software, our translation does not depend on the system used. We have implemented our approach using *Smodels* and *DLV*. We present several experimental comparisons between these systems and the *SHOP* planning system.
- We demonstrate that our method outperforms the transformation of a classical (i.e., STRIPS-style) planning problem into ASP proposed in [Son *et al.*, 2001] by a factor of 40-100. The particular relevance of that transformation method to our work is that, in their work, [Son *et al.*, 2001] proposed to use a form of control knowledge to speed up the classical planning process. In this paper, we show that HTN control knowledge provides more time-efficient transformations compared to the control strategies presented in [Son *et al.*, 2001].
- We investigate on how grounding affects the performance. It seems that systems allowing for unbound variables (without grounding) are better suited and would come closer in performance to *SHOP* than current ASP systems.

We have created a website where all our formalisations can be downloaded in a form ready to run on *DLV* and *Smodels*: `<http://www.cs.umd.edu/users/ukuter/ASP_Planning/>`. This site will be maintained and new examples will be added as we progress in our research.

### 1.1 Organisation

This paper is organised as follows. In the next section, we present the approaches in the literature which, we believe, are directly related to our efforts. In Section 3, we describe the HTN planning paradigm and the *SHOP* planning system. In Section 4, we present our causal theory for HTN planning and our translation method for transforming HTN planning problems into logic programs with negation. Section 5 contains our results. Our main theorem is that our translation method is correct and complete with respect to *SHOP*. We also present a variety of experimental results along with some discussions on the sources of complexity. In particular, we compare the performance of *DLV* and *Smodels* on planning benchmarks. Finally, we conclude with Section 6 and provide our future research directions.

## 2 Related Work

The published literature includes many efforts at formulating actions in logic programs and solving planning problems by using formulations such as [Gelfond and Lifschitz, 1998; Turner, 1997; Lifschitz, 1999; 2002]. [Gelfond and Lifschitz, 1998] describes three different action description languages that formalise theories of actions. These languages provide means to implement that formalisms as logic programs to solve planning problems effectively and efficiently [Lifschitz, 1999; Giunchiglia and Lifschitz, 1998]. The $\mathcal{C}$ language consists of general templates to define actions that have preconditions and effects. [McCain and Turner, 1997] presents a language for causal theories. They have also developed a system called *Ccalc*, which is a model checker for the language of causal theories translated from propositions in the $\mathcal{C}$ action language using rewrite rules [McCain, 1999]. The idea in all these works is to represent 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 the work presented here.

[Eiter *et al.*, 2002] 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 that describes the agent's knowledge about the world. Knowledge states may be incomplete, compared to the actual states of the world. This language is implemented as a front-end to the *DLV* logic programming system. [Eiter *et al.*, 2003] describes a language $K^c$, which extends the language $K$ for dealing with the action costs during planning. In particular, the language $K^c$ can express planning problems with optimality criteria, such as computing the shortest or the least-cost plans.

[Baral *et al.*, 2002] 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—as described in [Pearl, 1988]—to solve the planning problems with uncertainty.

[Dimopoulos *et al.*, 1997] presents a framework for encoding planning problems in logic programs with negation-as-failure. In this work, the idea is almost the same as ours, that is, the models of the logic program correspond to the plans. However, this work incorporates ideas from planners such as

*GraphPlan* and *SATplan*, and it does not consider any sort of search-control knowledge in the logic-program encodings. In this respect, our approach is completely different.

[Son *et al.*, 2001] discusses solving planning programs by logic programs. The difference between this work and the one described in [Dimopoulos *et al.*, 1997] is that [Son *et al.*, 2001] incorporates domain-dependent control knowledge to improve the performance of the planning. In this respect the work is similar to HTN planning. However, the encoding is conceptually different from HTN planning: it exploits domain constraints to define the ordering relationships between the actions, and uses these constraints to prune the search for correct sequence of actions to solve a planning problem. This technique does not eliminate an action in a state if it is applicable in that state and it satisfies the input constraints, although that action is not part of any solution for the input planning problem. In HTN planning, on the other hand, the search-control knowledge eliminates actions from consideration for the states that the planner visits during its search, which provides better search control.

## 3 Hierarchical Task Network (HTN) Planning

HTN planning is like classical planning in that each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition. However, HTN planners differ from classical planners in what they plan for, and how they plan for it.

The purpose of an HTN planner is to produce a sequence of actions that perform some activity or *task*. The description of a planning domain includes a set of operators similar to those of , and also a set of *methods*, each of which is a prescription for how to decompose a task into its *subtasks* (smaller tasks). Within a domain, the description of a planning problem contains an initial state. Instead of a goal formula, however, there is a partially ordered set of tasks to accomplish.

Planning proceeds by decomposing tasks recursively into smaller and smaller subtasks, until *primitive tasks*, which can be performed directly using the planning operators, are reached. For each task, the planner chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks or the interactions among them prevent the plan from being feasible, the planning system will backtrack and try other methods.

HTN planning has been proved to be more expressive than classical [Erol *et al.*, 1996]. Moreover, HTN planning algorithms have been experimentally proved to be more efficient than their action-based counterparts. This is because the domain knowledge and the notion of decomposing a task network while satisfying the given constraints enable the planner to focus on a much smaller portion of the search space than is typically searched by procedures. Due to their ability to generate plans very efficiently, HTN planners are used in a large variety of real-world applications [Wilkins, 1990; Currie and Tate, 1991; Nau *et al.*, 2003].

## 3.1 HTN planning using Ordered Task Decomposition (OTD)

In this paper, we are interested in a special case of HTN planning, namely HTN planning with Ordered Task Decomposition (OTD). This special case was first introduced in the *SHOP* system [Nau *et al.*, 1999; 2000]. The difference between *SHOP* and most other HTN-planning algorithms is that *SHOP* plans for tasks in the same order that they will later be executed. Planning for tasks in the order that those tasks will be performed makes it possible to know the current state of the world at each step in the planning process, which reduces the complexity of reasoning by eliminating a great deal of uncertainty about the world. This makes it easy to incorporate substantial inferencing and reasoning power into the planning system, including the ability to call external programs and the ability to perform numeric computations.

In order to do planning in a given planning domain, *SHOP* needs to be given knowledge about that domain. *SHOP*'s knowledge base contains *operators* and *methods*. Each operator is a description of what needs to be done to accomplish some primitive task, and each method is a prescription for how to decompose some compound (abstract) task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable. More than one method may be applicable to the same task, in which case there will be more than one possible way to decompose that task. Given the next task to accomplish, the *SHOP* algorithm nondeterministically chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. The deterministic implementation of the *SHOP* algorithm uses depth-first backtracking: if the constraints on the subtasks prevent the plan from being feasible, then the implementation will backtrack and try other methods.

As an example, Figure 1 shows two methods for the task of travelling from one location to another: *travelling by air*, and *travelling by taxi*. Travelling by air involves the subtasks of purchasing a plane ticket, travelling to the local airport, flying to an airport close to our destination, and travelling from there to our destination. Travelling by taxi involves the subtasks of *calling a taxi*, *riding in it to the final destination*, and *paying the driver*.

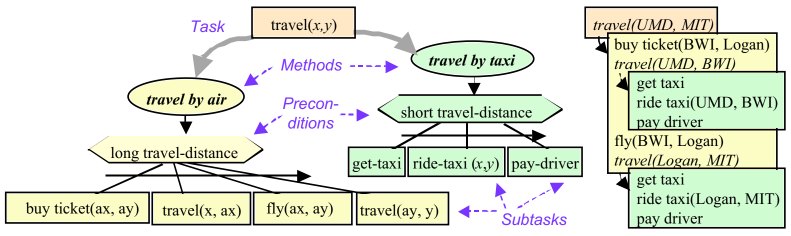


Figure 1. Travel planning example.

Note that each method's preconditions are not used to create subgoals (as would be done in ). Rather, they are used to determine whether or not the method is applicable: thus in Figure 1, the *travel by air* method is only applicable for long distances, and the *travel by taxi* method is only applicable for short distances. Now, consider the task of travelling from the University of Maryland to MIT. Since this is a long distance, the *travel by taxi* method is not applicable, so we must choose the *travel by air* method. As shown in Figure 1, this decomposes the task into the following subtasks: *(1)* purchase a ticket from Baltimore-Washington International (BWI) airport to Logan airport, *(2)* travel from the University of Maryland to BWI, *(3)* fly from BWI airport to Logan airport, and *(4)* travel from Logan airport to MIT. For the subtasks of travelling from the University of Maryland to BWI and travelling from Logan to MIT, we can use the *travel by taxi* method to produce additional subtasks as shown in Figure 1.

Here are some of the complications that can arise during the planning process:

- The planner may need to recognise and resolve interactions among the subtasks. For example, in planning how to travel to the airport, one needs to make sure one will arrive at the airport in time to catch the plane. To make the example in Figure 1 more realistic, such information would need to be specified as part of *SHOP*'s methods and operators.
- In the example in Figure 1, it was always obvious which method to use. But in general, more than one method may be applicable to a task. If it is not possible to solve the subtasks produced by one method, *SHOP* will backtrack and try another method instead.

## 3.2 HTN-planning with OTD: Syntax and Semantics

We use the same definitions for variable and constant symbols, predicate symbols, and terms, as in the *SHOP* planning system [Nau *et al.*, 1999;

2000]. Our definitions for logical atoms, states, tasks, task networks, axioms, operators, and methods are adapted from *SHOP*.

Following the notation used in *SHOP*, we will write logical atoms using the format (`name` $t_1 t_2 \dots t_n$), where `name` is a predicate symbol, and $t_1, t_2, \dots, t_n$ are terms. In *SHOP* we can classify the atoms into three kinds:

- *Rigid Atoms:* These are atoms whose truth values never change during planning. These atoms appear in states, but do not appear in the effects of planning operators nor in the heads of Horn clauses.
- *Primary Atoms:* These atoms can appear in states and in the effects of planning operators, but cannot appear in the heads of Horn clauses.
- *Secondary Atoms:* These are the ones whose truth values are inferred rather than being stated explicitly. They can appear in the heads of Horn clauses, but cannot appear in states nor in the effects of planning operators.

Now, we define the states and the axioms as in *SHOP*:

DEFINITION 1 (**States ($\mathcal{S}$), Axioms ($\mathcal{AX}$)**). A *state* $\mathcal{S}$ is a set of ground primary atoms. An *axiom* is an expression of the form

$$a \leftarrow l_1, \dots, l_n,$$

where $a$ is a secondary atom and the $l_1, \dots, l_n$ are literals that constitute either primary or secondary atoms.

Axioms need not be ground. We assume that the set of axioms does not contain cycles through negation.[1]

*SHOP* starts with a state $\mathcal{S}$ and modifies this state by taking into account the delete and add lists of the operators in the plan. Axioms are used only to check whether the preconditions of methods are satisfied. A precondition might not be explicitly satisfied (in the sense that the atom in question is contained in $\mathcal{S}$), but might be *caused by* $\mathcal{S}$ and the axioms $\mathcal{AX}$. The precise definition of this relation "*caused by*" is given as follows and extended in Subsection 4.

DEFINITION 2 (Literal caused by ($\mathcal{S}$,$\mathcal{AX}$)). A literal $l$ is caused by ($\mathcal{S}, \mathcal{AX}$) if $l$ is true in all answer sets of $\mathcal{S} \cup \mathcal{AX}$.

Because of our assumption on $\mathcal{AX}$, the set of axioms constitutes a stratified logic program which has exactly one answer set. This ensures that any

[1]This is just to ensure that a unique stable model always exist and thus the state is always complete (see the next definition). Without this condition, our approach is still complete but no more correct wrt. *SHOP*: *SHOP* does not terminate while our translation still gets meaningful results.

state described by the stable model of $\mathcal{S} \cup \mathcal{AX}$ is complete: any literal is either caused or its negation is caused.

In order to check which literals follow from $(\mathcal{S}, \mathcal{AX})$, *SHOP* uses an axiomatic inference procedure. To discuss this procedure, we need to make a distinction between the abstract *SHOP* algorithm and the *SHOP* implementation. On one hand, the abstract *SHOP* algorithm is nondeterministic and it makes no commitment to what inference procedure is used for checking whether literals follow from $(\mathcal{S}, \mathcal{AX})$. The completeness proof for *SHOP* [Nau *et al.*, 2000] says that if the inference procedure is complete, then *SHOP* is complete (i.e., if a planning problem has a solution, then at least one of *SHOP*'s execution traces will find a solution).

On the other hand, the *SHOP* implementation uses an inference procedure that does a depth-first search similar to the one in Prolog. This inference procedure is complete only if the axioms satisfy some restrictions similar to those needed in Prolog (no positive cycles, no cycles through negation).[2] However, all the axioms $\mathcal{AX}$ we are dealing with in this paper are of this sort. In fact, checking causality for these simple instances can be done in linear time.

A *task* is an expression of the form (`name` $t_1 t_2 \dots t_n$), where `name` (the task's name) is a task symbol, and $t_1, t_2, \dots, t_n$ (the task's arguments) are terms. A *ground task* is a task that has no variables in its arguments. A task can be either *primitive* (if it is to be accomplished directly in the world) or *compound* (if it is to be decomposed into other tasks). We use a prefix `?` to denote a variable (such as `?x` and `?y`) and `!` to denote the name of a primitive task. For example, to tell the planner that getting a taxi, riding in it, and paying the driver are primitive tasks, we would give them names like `!get-taxi`, `!ride-taxi`, and `!pay-driver`. Tasks using these names are `(!get-taxi ?x)`, `(!ride-taxi ?x ?y)`, or `(!pay-driver ?x ?y)`.

A *task list* is a list of tasks, like the following:

```
((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
```

A *ground task list* is a task list that consists of only ground tasks, like the following:

```
((!get-taxi umd) (!ride-taxi umd mit) (!pay-driver umd mit)))
```

An operator specifies how to accomplish a primitive task by modifying the current state of the world by removing every atom in its delete list and by adding every atom in its add list.

[2]In addition, the *SHOP* implementation also computes its task decompositions using a depth-first search. Thus, in order to achieve completeness, the HTN methods also need to satisfy a similar acyclicity restriction.

DEFINITION 3 (**Operator: (Op** $h$ $\epsilon_{del}$ $\epsilon_{add}$**)** ). An *operator* is an expression of the form **(Op** $h$ $\epsilon_{del}$ $\epsilon_{add}$**)**, where $h$ (the *head*) is a primitive task and $\epsilon_{add}$ and $\epsilon_{del}$ are lists of primary atoms (called the *add*- and *delete-lists*, respectively). The set of variables in the atoms in $\epsilon_{add}$ and $\epsilon_{del}$ must be a subset of the set of variables in $h$.[3]

As an example, here is a possible implementation of the `get-taxi` operator from Figure 1:

```
(:Op (!get-taxi ?x)
    ((taxi-called-to ?x))
    ((taxi-standing-at ?x)))
```

Operators are used in decomposition of primitive tasks during planning:

DEFINITION 4 (**Decomposition of Primitive Tasks**). Let $t$ be a primitive task, and let Op = **(Op** $h$ $\epsilon_{del}$ $\epsilon_{add}$**)** be an operator. Suppose that $\theta$ is a unifier for $h$ and $t$. Then the ground operator instance (Op)$\theta$ is *applicable* to $t$, in which case we define the *decomposition* of $t$ by Op to be (Op)$\theta$.

The decomposition of a primitive task by an operator results in a ground instance of that operator – i.e., it results in an action that can be applied in a state of world. We now define the result of such an application:

DEFINITION 5 (**Plans, result($\mathcal{S}$,$\pi$)**). A *plan* is a list of heads of ground operator instances.[4] A plan $\pi$ is called a *simple plan* if it consists of the head of just one ground operator instance.

Given a simple plan $\pi = (h)$, we define result$(\mathcal{S}, \pi)$ to be the set

$$\mathcal{S} \setminus \epsilon_{del} \cup \epsilon_{add}$$

obtained by deleting from $\mathcal{S}$ all atoms in $\epsilon_{del}$ and by adding all ground instances of atoms in $\epsilon_{add}$.

If $\pi = (h_1, h_2, \ldots, h_n)$ is a plan and $\mathcal{S}$ is a state, then the *result* of applying $\pi$ to $\mathcal{S}$ is the state result$(\mathcal{S}, \pi)$ = result(result($\ldots$(result $(\mathcal{S}, h_1), h_2), \ldots), h_n)$.

In *SHOP*, a method specifies a possible way to accomplish a compound task. The set of methods relevant for a particular compound task can be seen as a recursive definition of that task.

[3]Unlike the operators used in , ours have no preconditions. Preconditions are not needed for operators in our formulation, because they occur in the methods that invoke the operators.

[4]In Definition 8, we will require that in any planning domain, every planning operator must have a unique name. This is sufficient to guarantee that every plan specifies an unambiguous sequence of operator instances.

DEFINITION 6 (**Method: (Meth** $h$ $\rho$ **t)** ). A *method* is an expression of the form **(Meth** $h$ $\rho$ **t)** where $h$ (the method's *head*) is a compound task, $\rho$ (the method's *preconditions*) is a conjunction of literals and **t** is a totally-ordered list of subtasks, called the *decomposition list* of the method. The set of variables that appear in the decomposition list of a method must be a subset of the variables in $h$ (the head of the method) and $\rho$ (the preconditions of the method). [5]

Here is a possible implementation of the `travel-by-taxi` method from the same figure:

```
(:Meth (travel ?x ?y)
    ((smaller-distance ?x ?y))
    ((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
```

Let $m =$ **(Meth** $h$ $\rho$ **t)** be a method. Note that there may be variables in $\rho$ that do not appear in the head $h$ of the method $m$. These variables are called the *unbound variables* of $m$. During planning, these variables are grounded when the method is used for the decomposition of a compound task, as described below.

DEFINITION 7 (**Decomposition of Compound Tasks**). Let $t$ be a compound task, $\mathcal{S}$ be the current state, $Meth =$ **(Meth** $h$ $\rho$ **t)** be a method, and $\mathcal{AX}$ be an axiom set. Suppose that $\theta$ is a unifier for $h$ and $t$, and that $\theta'$ is a unifier such that all literals in $(\rho)\theta\theta'$ are caused wrt. $\mathcal{S}$ and $\mathcal{AX}$ (see Definition 2).

Then, the ground method instance $(Meth)\theta\theta'$ is *applicable* to $t$ in $\mathcal{S}$, and the result of applying it to $t$ is the ground task list $\mathbf{r} = (\mathbf{t})\theta\theta'$. The task list $\mathbf{r}$ is the *decomposition* of $t$ by *Meth* in $\mathcal{S}$.

Note that the decomposition of a compound task by a method does not change the state of the world. The result of such a decomposition is a ground task list that needs to be further decomposed until we get a list of only ground operator instances — i.e., a plan.

DEFINITION 8 (**Planning Domain Descriptions and Problems**). A *planning domain description* $\mathcal{D}$ is a triple consisting of (1) a set of axioms, (2) a set of operators such that no two operators have the same head, and (3) a set of methods.

[5]This restriction is needed to ensure that our programs do not violate the safeness restrictions of the ASP systems we are using. However, the restriction has no effect on the expressivity of our formalism. Any method that does not satisfy the restriction can easily be translated into an equivalent method that does satisfy the restriction, by introducing a dummy precondition that can always be satisfied.

A *planning problem* is a triple $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is a state, $\mathbf{t}= (t_1, t_2, \ldots, t_k)$ is a ground task list, and $\mathcal{D}$ is a planning domain description.

We now define a *solution* of a planning problem.

DEFINITION 9 (**Solutions**). Let $P = (\mathcal{S}, \mathbf{t}, \mathcal{D})$ be a planning problem and $\pi = (h_1, h_2, \ldots, h_n)$ be a plan. Then, $\pi$ is a *solution* for $P$,[6] if any of the following is true:

- Case 1: $\mathbf{t}$ and $\pi$ are both empty, (i.e., $k = 0$ and $n = 0$);
- Case 2: $\mathbf{t} = (t_1, t_2, \ldots, t_k)$, $t_1$ is a ground primitive task, $(h_1)$ is the decomposition of $t_1$, and $(h_2 \ldots h_n)$ solves $(\text{result}(\mathcal{S}, (h_1)), (t_2, \ldots, t_k), \mathcal{D})$;
- Case 3: $\mathbf{t} = (t_1, t_2, \ldots, t_k)$, $t_1$ is a ground compound task, and there is a decomposition $(r_1 \ldots r_j)$ of $t_1$ in $\mathcal{S}$ such that $\pi$ solves $(\mathcal{S}, (r_1, \ldots, r_j, t_2, \ldots, t_k), \mathcal{D})$.

The planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is *solvable* if there is a plan that solves it.

One important issue that we want to point out about this definition is that the *SHOP* formalism does not require the tasks to be ground. This and the restriction in Definition 6, are both necessary in the formalism of our translation method, simply because, otherwise, the logic programs that are generated by our translation would contain rules that violate the safeness conditions that are imposed by current ASP systems. However, this is a mild restriction and can always be ensured by adding dummy predicates.

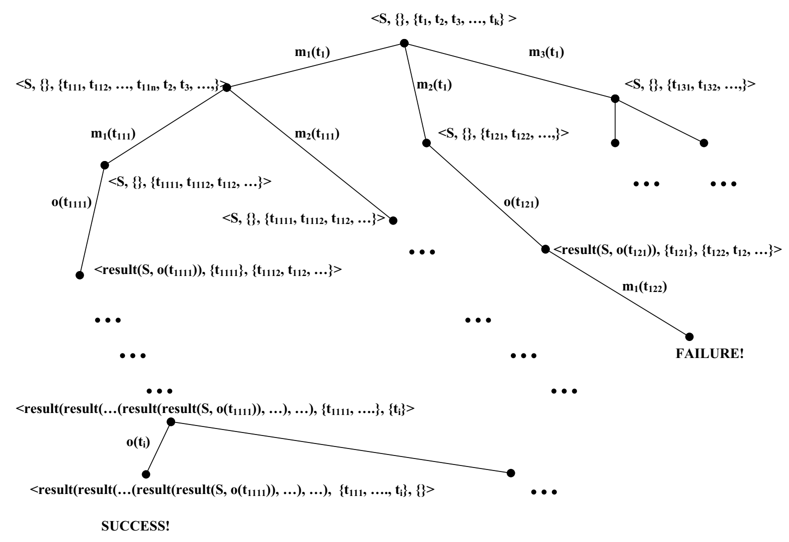
It will be very helpful for the main proof of Theorem 30 to introduce the notion of a *search ree*. The successful paths of this tree correspond to the solutions of the planning problem.

DEFINITION 10 (Search Tree for $\mathfrak{Trans}(\cdot)$).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, we define the search tree for $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ as follows. Nodes of the tree are triples of the form $\langle \mathcal{S}', \mathbf{t}_{caused}, \mathbf{t}' \rangle$, where $\mathcal{S}'$ is a state, $\mathbf{t}_{caused}$ is an ordered list of ground primitive tasks, and $\mathbf{t}'$ is a (possibly empty) ordered list of ground (compound or primitive) tasks.

The start node consists of the triple $\langle \mathcal{S}, \emptyset, \mathbf{t} \rangle$. Leaf nodes are those of the form $\langle \mathcal{S}, \mathbf{t}_{caused}, \emptyset \rangle$. Branches ending in such leaves are called *successful.* Given a node $\langle \mathcal{S}', \mathbf{t}_{caused}, \mathbf{t}' \rangle$ with $\mathbf{t}' \neq \emptyset$, its children are defined as follows:

- If the first task in $\mathbf{t}'$ is primitive and there is an operator in $\mathcal{D}$ for it, then there is exactly one child $\langle \mathcal{S}^\star, \mathbf{t}^\star_{caused}, \mathbf{t}^\star \rangle$. $\mathbf{t}^\star_{caused}$ is the old list $\mathbf{t}_{caused}$ plus this first task appended. $\mathcal{S}^\star$ is obtained by modifying $\mathcal{S}'$ according to the add and delete lists of the operator. $\mathbf{t}^\star$ is $\mathbf{t}'$ with the first element

[6]Or equivalently, we say that $\pi$ *solves* $P$, or $\pi$ *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).



<S, {}, {t1, t2, t3, ..., tk} >
m1(t1)
m2(t1)
m3(t1)
<S, {}, {t111, t112, ..., t11n, t2, t3, ...}>
<S, {}, {t121, t122, ...}>
<S, {}, {t131, t132, ...}>
m1(t111)
m2(t111)
<S, {}, {t1111, t1112, t112, ...}>
<S, {}, {t1111, t1112, t112, ...}>
o(t1111)
o(t121)
<result(S, o(t1111)), {t1111}, {t1112, t112, ...}>
<result(S, o(t121)), {t121}, {t122, t12, ...}>
m1(t122)
FAILURE!
<result(result(...(result(result(S, o(t1111)), ...), ...), {t1111, ....}, {tl}>
o(tl)
<result(result(...(result(result(S, o(t1111)), ...), ...), {t111, ...., tl}, {}>
SUCCESS!

Figure 2. Search Tree for $(\mathcal{S}, \mathbf{t}, \mathcal{D})$. Edge labellings $m_i(t)$ (resp. $o(t)$) represent a method (resp. an operator) application to a task $t$, which is compound (resp. primitive).

deleted. The edge to this child is labelled with the name of this first task. If the first task in $\mathbf{t}'$ is primitive and there is no operator, then there is no child and the branch is marked with *Failure*.

- If the first task in $\mathbf{t}'$ is compound, and there exist method instances applicable to it (according to Definition 7), then each such method instance $m_i$ leads to a child node $\langle \mathcal{S}, \mathbf{t}_{caused}, \mathbf{t}^i \rangle$ the edge to which is labelled with $m_i$. $\mathbf{t}^i$ is obtained from $\mathbf{t}'$ by replacing the first task by the subtasks according to $m_i$. If the first task in $\mathbf{t}'$ is compound, and there are no methods applicable to it, then the branch is marked *Failure*.

We define the *task-depth* of a node and its edges as follows. The start node gets task-depth 0. Whenever a method is used to extend a node, the children nodes keep the same task-depth. When an operator is applied, and thus a task is moved from $\mathbf{t}'$ into $\mathbf{t}_{caused}$ then the task depth of the child node is incremented by one. Obviously, the task depth of a node is the size of the list of tasks in $\mathbf{t}_{caused}$.

Such a tree (or a part thereof) is depicted in Figure 2. Note that there can be different paths (corresponding to the application of different methods) that finally lead to the same plans (as a list of the heads of ground instances of operators).

DEFINITION 11 (Solution Set of a Planning Problem: $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$). Let $P = (\mathcal{S}, \mathbf{t}, \mathcal{D})$ be a planning problem, and suppose $\mathcal{T}$ is the search tree for $P$. Then, $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is a *multi set*: it contains exactly the ordered lists $\mathbf{t}_{caused}$ in the leaf nodes that are reached by the successful paths of $\mathcal{T}$.

We also say $\mathcal{T}$ represents $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$. Note that $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ may contain more than one copy of the same plan.

## 4 Encoding HTN planning in Nonmonotonic Logic Programming

Our approach of encoding HTN planning problems as logic programs is based on *SHOP*'s representation of a planning problem as described in the last section. We now present the first steps of a causal theory of HTN planning based on that formalism. This theory serves as an intermediate step and a motivation for our translation methodology, which is given in the next subsection. We conclude this section with the formalisation of a particular example.

### 4.1 Causal Theory for HTN Planning

In this section we prepare the ground for our translation in the next subsection. We give some definitions of a causal theory for HTN planning in a *SHOP*-like ordered task decomposition.

DEFINITION 12 (**Causable Literals**). Let $\mathcal{S}$ be a state, and let $\mathcal{D}$ be a planning domain description. A literal $l$ is causable wrt. $(\mathcal{S},\mathcal{D})$ if it is caused by $(\mathcal{S}, \mathcal{AX})$ (according to Definition 2), where $\mathcal{AX}$ is the set of axioms in $\mathcal{D}$.

A conjunction of literals is causable wrt. $(\mathcal{S}, \mathcal{D})$ if every literal in the conjunct is causable wrt. $(\mathcal{S}, \mathcal{D})$ (according to the Definition 2).

DEFINITION 13 (**Causable Tasks**). Let $\mathcal{S}$ be a state, and let $\mathcal{D}$ be a planning domian description. The definition of an ordered list of ground tasks to be causable wrt. $(\mathcal{S}, \mathcal{D})$ comes in three steps.

1. The empty list [ ] is causable wrt. $(\mathcal{S}, \mathcal{D})$.

2. An ordered list of ground primitive tasks $t_1, \ldots, t_n$ is causable wrt. $(\mathcal{S}, \mathcal{D})$ if for each $t_i$, there exists an operator $(\mathbf{Op}\ h\ \epsilon_{del}\ \epsilon_{add}) \in \mathcal{D}$ and there is a unifier $\theta$ such that $t_i = (h)\theta$.

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

- There exists a method $(\textbf{Meth}\ h\ \rho\ \{t_{j_1}, \dots, t_{j_m}\}) \in \mathcal{D}$ for $t_j$, and there is a unifier $\theta$ such that $t_j = (h)\theta$; and
- There exists a unifier $\theta'$ such that the preconditions list $(\rho)\theta\theta'$ is causable wrt. $(\text{result}(\mathcal{S}, (t_1, \dots t_{j-1})), \mathcal{D})$; and
- The ordered decomposition list

  $$(t_1, \dots, t_{j-1}, (t_{j_1})\theta\theta', \dots, (t_{j_m})\theta\theta', t_{j+1}, \dots t_n)$$

  is causable wrt. $(\mathcal{S}, \mathcal{D})$.

Note that this is a recursive definition. The condition in the last part (*compound tasks*) eventually ends when there are only primitive tasks left, and thus the second part (*primitive tasks*) can be applied. The notion of *literals being causable* is used to make sure that the appropriate methods (used to decompose the task $t_j$) can be applied in the current state.

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 in the subsequent section. The next theorem states the equivalence of the original *SHOP* planning framework as presented in the last section with the notion of causable tasks just introduced.

THEOREM 14.
*Let $P = (\mathcal{S}, \mathbf{t}, \mathcal{D})$ be a planning problem. Then, there is a solution to $P$* if and only if *the task list* $\mathbf{t}$ *is causable wrt. $(\mathcal{S}, \mathcal{D})$.*

**Proof.** Rather than giving a full proof using structural induction, we give a detailed proof sketch from which the full proof can be easily worked out.

The proof starts by recursively constructing the solution of an HTN planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ and showing the causal relationships based on our causal theory at the same time.

Suppose there exists a solution to $(\mathcal{S}, \mathbf{t}, \mathcal{D})$. If $\mathbf{t}$ is empty, then $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ contains exactly one plan, namely the empty plan. This is because of the fact that there will be no tasks to be accomplished—thus, no task to be causable . If $\mathbf{t}$ is not empty, and consists of primitive tasks only, then there must be operators for all these tasks and thus, by Definition 13 (2nd step), $\mathbf{t}$ is causable wrt. $(\mathcal{S}, \mathcal{D})$.

We now reduce the general case to the case where only primitive tasks occur. Let $\mathbf{t}$ be non-empty, and assume it contains compound tasks. Then we recursively carry out the task decompositions (see Definition 7) until we reach a list of ground primitive tasks. This is possible because a solution exists (this solution is given by the final list of primitive tasks). Note that there might exist different such lists, corresponding to different choices of decompositions via methods. But this list is causable according to the second part of Definition 13 (*primitive tasks*).

Now, according to the third part of Definition 13, we can recursively replace the primitive tasks with the compound tasks they were induced from (via methods) and we get the result that all the intermediate ordered lists obtained in that way themselves are causable wrt. $(\mathcal{S}, \mathcal{D})$. Note that the conditions in the third part of Definition 13 correspond exactly to the notion of the decomposition of a compound task (Definition 7).

Therefore, it follows from the recursive construction above that if a list of ground tasks $\mathbf{t}$ is achieved according to our planning theory, it must be causable as well.

The proof of the converse is similar. Once a list of ground tasks $\mathbf{t}$ is causable wrt. $(\mathcal{S}, \mathcal{D})$, we can find a list of primitive tasks that is causable. This list is obtained by certain decompositions, and these decompositions constitute a solution of the planning problem. ■

## 4.2 Encoding Planning Problems as Logic Programs

In this section, we present our translation method for encoding planning problems as logic programs with ASP semantics. Our translation method is a general technique that is independent from the implementation details and syntactic requirements of the any underlying ASP system. Note that there are several differences between the syntactic requirements of the ASP systems. In this respect, the presentation in this section is given in a more conceptual level in general; however, where necessary, we adapted the syntax of *Smodels*.[7]

Translating a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ to its logic program counterpart requires encoding the initial state and the state transition characteristics of *SHOP*, the goal tasks and the ordered task decomposition technique in *SHOP*, and the domain description including the axioms, the operators, and the methods, which are given in the description of a planning problem. For this reason, we describe our translation method in several steps such that each step encodes a part of the complete translation corresponding to the components of a planning problem as described above. Combining these steps yield a complete logic program in ASP semantics that is capable of

[7]However, note that when implementing our translation methodology, one must address the syntax requirements of the underlying system that is being used. In this paper, we concentrated on the two ASP systems *Smodels* and *DLV*, so we made system-specific syntactic changes to the conceptual description of our translation method during the implementation in these systems. Our complete implementation of planning examples for both *DLV* and *Smodels* are available at `<http://www.cs.umd.edu/users/ukuter/ASP_Planning/>`.

solving planning problems in the way that *SHOP* does.

We now present our main definition:

DEFINITION 15 ($\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$: Translation for the Planning Problem). Let $P = (\mathcal{S},\mathbf{t},\mathcal{D})$ be a planning problem. The logic program $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$ that solves $P$ is defined as

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

where

- $\mathfrak{Trans}(\perp)$ is the logic program segment that marks the successful termination of the planning process,
- $\mathfrak{Trans}(\mathcal{S})$ is the logic program segment that encodes the initial state $\mathcal{S}$,
- $\mathfrak{Trans}(\mathbf{t})$ is the logic program segment that encodes the goal task list $\mathbf{t}$,
- $\mathfrak{Trans}(\mathcal{AX})$ is the logic program segment that encodes the axioms given in the domain description $\mathcal{D}$,
- $\mathfrak{Trans}(\mathcal{OP})$ is the logic program segment that encodes the operator descriptions given in $\mathcal{D}$, and
- $\mathfrak{Trans}(\mathbf{F})$ is the logic program segment that encodes the state-transition characteristics of *SHOP*, and
- $\mathfrak{Trans}(\mathcal{METH})$ is the logic program segment that encodes the method descriptions given in $\mathcal{D}$.

In the following subsections, we give the definitions for the logic program segments mentioned above.

**The Time Line**

In order to be able to keep track of different states of the world in an answer set of a logic program, we attached a *time variable* that will occur in many predicates in our translation. The domain of a time variable is a *time line*, which is a set of integers $\{0, 1, \ldots, \tau\}$, where $\tau$ is called the end point of the time line.

Given a planning problem $P = (\mathcal{S},\mathbf{t},\mathcal{D})$, we need to know $\tau$ in advance in order for our translation methodology to work. $P$ does not specify this information since *SHOP* does not need a notion of time line during planning — the search process naturally differentiates different states of the world. One way to determine a correct value for $\tau$ is an incremental approach. That is, we start with $\tau = 0$, use our translation to produce the logic program $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$, and generate the answer sets of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$. Then, we increment $\tau$, and repeat the whole process until no new answer sets are generated by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$. Note that, using this approach, we eventually generate all of the possible answer sets of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, as shown in Theorem 30.

An alternative approach would be as follows. Let $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be the set of solutions of $P$, as in Definition 11. Then, we define $\tau$ as

$$\tau = max\{|\pi| \ : \ \pi \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})\},$$

where $|\pi|$ denotes the size of the solution $\pi$. Note that we can find the set $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ by solving $P$ using *SHOP*.

We use time variables in various rules in our translation, so before going into the details of the translation, we believe that the following points are worth noting:

1. As described above, we assume that planning starts at time point 0 (see Definitions 16 and 19).

2. Planning proceeds by selecting a task to be accomplished next (see Definition 19. The rules about *taskTBA* and *causable* are given in Definition 17). Note that the task to be decomposed may be either primitive or compound.

3. The time variable $T$ is incremented only when the task to be decomposed is a primitive task and there is an operator for it (a simple reduction) in the domain description provided as a part of the planning problem (see Definition 18).

   This means that, in general, there may be more than one task that is selected and decomposed at a particular time point $T$. However, among these tasks, there is only one primitive task at any particular point in time. For example, consider Figure 2. Task $t_1$ is a compound task and so are $t_{111}$ and $t_{1111}$ (obtained by respective methods). So $taskTBA(t_1, 0), taskTBA(t_{111}, 0)$, $taskTBA(t_{1111}, 0)$ are all true (resp. hold in a stable model). Only after the primitive task $t_{1111}$ has been accomplished by an operator is the time incremented by 1.

4. As a result of this formulation, the task depth in the search tree corresponds to the value of the time variable $T$.

## Encoding the Initial State

*SHOP*'s initial state is a set of ground atoms. In this respect, given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the logic program encoding for the initial state $\mathcal{S}$ is defined as follows:

DEFINITION 16 ($\mathfrak{Trans}(\mathcal{S})$: Translation for Initial State).
Given a planning problem $(\mathcal{S}, \mathtt{t}, \mathcal{D})$, for each ground atom $a \in \mathcal{S}$, the logic program $\mathfrak{Trans}(\mathcal{S})$ contains the rule

$$in\_state(a, 0) :-$$

where 0 indicates the initial time.

**Encoding the Goal Task(s)**

Given a planning problem $(\mathcal{S}, \mathtt{t}, \mathcal{D})$, where $\mathcal{S}$ is the initial state, $\mathtt{t}$ is a ground task list, and $\mathcal{D}$ is a domain description for this planning problem, the aim of the planning process is to find a plan that accomplishes all of the (goal) tasks in $\mathtt{t}$ from the initial state $\mathcal{S}$ in the order they are given (according to Definition 8). 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, and this is due to the Definition 13 and a direct consequence of Theorem 14.

In this respect, planning proceeds by selecting a task as the "current task" – i.e., the task that the planner will try to accomplish next. In the logic programs produced by our translation, this is encoded by using a special predicate defined as follows:

DEFINITION 17 (Tasks To Be Accomplished). Given a planning problem $(\mathcal{S}, \mathtt{t}, \mathcal{D})$, we define a special predicate $taskTBA\_n$ for each possible task (e.g. primitive or compound) such that if the task that needs to be decomposed at time $T$ is $h \equiv (name_h \ arg_1 \ arg_2 \ \ldots \ arg_N)$ then $taskTBA\_n(name_h, arg_1, arg_2, \ldots, arg_N, T)$ denotes this fact and $n$ is a natural number which equals $N + 2$ ($n$ is the number of arguments of this predicate).

As an example, if the task to be accomplished is travelling from UMD to MIT denoted as (`travel` *umd mit*), then we use the predicate $taskTBA\_4(travel, umd, mit, T)$ to denote this fact. For the sake of clarity, we will use the shorthand notation $taskTBA(h, T)$ in the rest of the paper.

We define the fact that whether a task is causable as follows:

DEFINITION 18 (CAUSABLE). Given a ground task $t$, we define

$CAUSABLE(t, T_s, T_a)$ as follows:

$$\begin{cases} false & \text{if } t \text{ is a primitive task and there is no operator for it in } \mathcal{D}, \\ false & \text{if } t \text{ is a compound task and there is no method for it in } \mathcal{D}, \\ taskTBA(t, T_s) \text{ and } T_a = T_s + 1 & \text{if } t \text{ is a primitive task and there is an operator for it in } \mathcal{D}, \\ causable(t, T_s, T_a) & \text{if } t \text{ is a compound task and there is a method for it in } \mathcal{D}. \end{cases}$$

where $T_s$ denotes the time when the task $t$ was selected to be decomposed and $T_a$ denotes the time when $t$ is actually accomplished (i.e., $T_a$ is the time when $t$ is caused).

In the definition above, $causable(t, T_s, T_a)$ is a shorthand notation for the predicate $causable\_n(name_t, arg_1, arg_2, \ldots, arg_N, T_s, T_a)$ in which the symbol $n = N+3$ denotes the number of arguments of the predicate $causable\_n$. For the sake of clarity, we will use $causable(t, T_s, T_a)$ in the rest of the paper.

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

DEFINITION 19 ($\mathfrak{Trans}(\mathbf{t})$: 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 ground tasks. Then, $\mathfrak{Trans}(\mathbf{t})$ is the logic program that contains one rule for each ground task $h_i$, where $i = 1, 2, \ldots, n$, as follows:

1. Case 1: $i = 1$,
$$taskTBA(h_1, 0) \ :-$$

2. Case 2: Otherwise,
$$taskTBA(h_i, T_i) \quad :- \quad CAUSABLE(h_{i-1}, T_{i-1}, T_i),\ T_i \geq T_{i-1}.$$

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 19 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 14.

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 problem. The following definition is given to encode the successful termination of the planning process.

DEFINITION 20 ($\mathfrak{Trans}(\bot)$: Successful Termination). Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the logic program segment $\mathfrak{Trans}(\bot)$ 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:

$$\begin{array}{lll} plan\_found & :- & CAUSABLE(h_n, T_n, T_{n+1}), T_{n+1} \geq T_n. \\ plan\_found & :- & not\ plan\_found. \end{array}$$

where $h_n$ is the last goal task in $\mathbf{t}$, $T_n$ denotes the time at $h_n$ is decomposed by a simple reduction for it (see Definition 7), and $T_{n+1}$ is the time at which $h_n$ is causable (accomplished).

These two rules together state that if the last goal task is causable then there is a plan (solution) for the planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ as a result of Definition 19. Otherwise, there is none.

**Encoding the Axioms**

We now define the logic program segment that encodes the axioms of a domain description. We start with notion of *translation* for a literal.

DEFINITION 21 (Translation for Literals).
Given a literal, $l$, we define $C(l, T)$, the *translation* of $l$ at time $T$, as

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

DEFINITION 22 ($\mathfrak{Trans}(\mathcal{AX})$: Translation for Axioms).
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, $\mathfrak{Trans}(\mathcal{AX})$ is the logic program segment that contains the following rules: for all " $a \leftarrow l_1, \ldots, l_n$ " $\in \mathcal{AX}$,

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

where $C(l_i, T)$ is the translation of the literal $l_i$, as defined in Definition 21 above.

**Encoding the Operators and the State Transitions**

*SHOP* uses the operator descriptions in $\mathcal{D}$ in decomposition of primitive tasks that needs to be accomplished during planning. In the translation, the logic program segment that encodes the operators in $\mathcal{D}$ is given as follows:

DEFINITION 23 ($\mathfrak{Trans}(\mathcal{OP})$: 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 that contains the following rules: for all $a \in \mathrm{Del}(\mathsf{Op})$:

$$out\_state(a, T+1)\ :-\ taskTBA(h, T).$$

and for all $a \in \mathrm{Add}(\mathsf{Op})$:

$$in\_state(a, T+1) \ :- \ taskTBA(h, T).$$

where $h$ is a primitive task – i.e., the ground head of the operator that is used in the decomposition of $h$.

Note that these two rules encode the delete- and the add-lists of the operator respectively. The result of a decomposition of a primitive task by an operator application is a new state of the world, which is generated by deleting all of the atoms that are in the delete-list of that operator from the current state and by adding all of the atoms that are in the add-list of that operator to the current state.

An operator only describes the *change* it causes to occur in the current state. The planner is still responsible for keeping track of the other facts that remain *unchanged* after an operator application. This is known as the famous *Frame Problem* in AI Planning. In the translation, the logic program segment that addresses the frame problem is defined as follows:

DEFINITION 24 ($\mathfrak{Trans}(\mathbf{F})$: Keeping Track of the State $\mathcal{S}$).
The logic program segment $\mathfrak{Trans}(\mathbf{F})$ that encodes the frame axiom is defined as follows:

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

Note that the state of the world in *SHOP* consists of only positive ground primary atoms. Basically the above rule states that if a positive ground atom, $a$, is initially true, then it should be true in the next state unless it has been marked as to be deleted during the transition from the current state to the next state.

**Encoding the Methods**

*SHOP* uses the method descriptions in $\mathcal{D}$ in decompositions of compound tasks that need to be accomplished during planning. Before proceeding with the definition of the logic program segment that encodes the methods in $\mathcal{D}$, we give the following definition:

DEFINITION 25 (Methods for a Compound Task). Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ and a compound task $h \equiv (name_h \ arg_1 \ arg_2 \ \ldots \ arg_N)$, we define a special predicate $method\_n\_i(name_h, arg_1, arg_2, \ldots, arg_N, T)$ for each method $m_i \in \mathcal{D}$ whose head unifies with $h$. The symbol $n$ in the predicate name denotes the number of arguments of the predicate, i.e. $n = N+2$.

For purposes of clarity, we will use the following shorthand notations in the rest of this paper: Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ and a compound task $h$, if there is only one method $m$ whose head unifies with

$h$ in $\mathcal{D}$ then we will use $method(h,T)$ to refer to $m$, instead of the $method\_n\_1(name_h, arg_1, arg_2, \ldots, arg_N, T)$ predicate as defined above. If there are more than one methods $m_i$ for $h$ then we will use the notation $method_i(h,T)$ for each such method $m_i$.

We present the translation for methods of *SHOP* in four steps: (1) the translation for the nondeterministic choice of applying alternative methods to a particular compound task; (2) the translation for evaluating the preconditions of a method in order to decide whether it is applicable to a particular compound task; (3) the translation for the task decomposition specified by a method; and (4) the translation for the accomplishment of a particular task by a method. Given a *SHOP* method, if the translations in these four steps are performed, then we produce a logic program segment, $\mathfrak{Trans}(\mathcal{METH})$, which is the ASP encoding of that method.

DEFINITION 26 (Translation for Encoding Alternative Methods). Given a planning problem $P = (\mathcal{S}, \mathtt{t}, \mathcal{D})$, let $h$ be a compound task that needs to be accomplished in the solution of $P$. Suppose $\mathcal{D}$ contains $N$ methods whose heads unify with $h$; namely, $m_1, m_2, \ldots, m_N$. Then, the logic program segment that encodes the nondeterministic choice of which method to apply to the task $h$ is as follows: for $i, j = 1, \ldots, N$,

$$\begin{array}{lll} method_i(h,T) & :- & \bigwedge_{i \neq j} not\ method_j(h,T), \\ & & taskTBA(h,T). \end{array}$$

Intuitively, the rules in Definition 26 enforce the translation to create $N$ different answer sets for $N$ possible method applications to the task $h$. This is due to the fact that each such method specifies a different way to decompose the task $h$, and therefore, the planner can find different solutions due to each such method.

Next, we present the translation for evaluating the preconditions of each such method. Note that if the precondition of the method is not satisfied in the state of the world, then the method cannot be applied to the task $h$.

DEFINITION 27 (Translation for Precondition Evaluations). Let $h$ be a compound task that needs to be accomplished, $m_i$ be a method whose head unifies with $h$, and $\rho$ be the preconditions of $m_i$. Then, we have two steps:

1. Let $p \in \rho$ be a precondition of $m$ and $\chi_1, \chi_2, \ldots, \chi_f$ be the unbound variables in $p$ such that if $f = 0$ then $p$ has no such variables. Suppose that $R_j$ denotes the *range* of $\chi_j$ in $p$ – i.e., $R_j$ is the set of all possible instantiations of $\chi_j$ in the world. Then, for each unbound variable $\chi_j$ of $p$, we create new variable symbols $\chi_{j,k}$ such that $j = 1, \ldots, f$ and $k = 1, \ldots, |R_j|$.

2. For each precondition $p \in \rho$,

   - if $p$ does not contain unbound variables, then we have

     $$\mathit{checked\_state}(p, T) \ :- \ C(p, T), \mathit{method}_i(h, T).$$

     where $C(p, T)$ is as defined in Definition 21.

   - Otherwise, we have

$$\begin{array}{l}
\mathit{checked\_state}(p(\chi_{1,1}, \chi_{2,1}, \ldots, \chi_{f,1}), T) \ :- \\
\qquad \mathit{method}_i(h, T), \ \mathit{in\_state}(p(\chi_{1,1}, \chi_{2,1}, \ldots, \chi_{f,1}), T), \\
\qquad \mathit{not}\, \mathit{checked\_state}(p(\chi_{1,1}, \chi_{2,1}, \ldots, \chi_{f,2}), T), \\
\qquad \vdots \\
\qquad \mathit{not}\, \mathit{checked\_state}(p(\chi_{1,1}, \chi_{2,1}, \ldots, \chi_{f,|R_f|}), T), \\
\qquad \vdots \\
\qquad \mathit{not}\, \mathit{checked\_state}(p(\chi_{1,1}, \chi_{2,|R_2|}, \ldots, \chi_{f,|R_f|}), T), \\
\qquad \mathit{not}\, \mathit{checked\_state}(p(\chi_{1,2}, \chi_{2,1}, \ldots, \chi_{f,1}), T), \\
\qquad \vdots \\
\qquad \mathit{not}\, \mathit{checked\_state}(p(\chi_{1,|R_1|}, \chi_{2,|R_2|}, \ldots, \chi_{f,|R_f|}), T), \\
\qquad \bigwedge_{j=1}^{f} \chi_{j,1} \ != \chi_{j,2} \ != \ldots \ != \chi_{j,|R_j|}.
\end{array}$$

     where $\chi_1, \chi_2, \ldots, \chi_f$ are the unbound variables in $p$.

Intuitively, the rules of Definition 27 create an answer set for each possible instantiation of the unbound variables of $m_i$ in the world. This is due to the fact that *SHOP* creates an instance of $m_i$ for each instantiation of the unbound variables in $m_i$, and decomposes the task $h$ with each such method instance. In order for our translation to be correct, we need to simulate this behavior of *SHOP* in our translation since ASP systems do not provide such semantics, to the best of our knowledge.

Note that for each precondition $p \in \rho$, there must be at least one answer set in which both $\mathit{method}_i(h, T)$ and $\mathit{checked\_state}(p, T)$ are true for the method $m_i$ to be applicable to the task $h$ in the current state of the world (denoted by the time variable $T$). If there is no such answer set then it means that $m_i$ is not applicable to $h$.

Now that we have established the rules for checking the applicability of $m_i$ to $h$, we are ready to give the definition for the decomposition of the task $h$ by $m_i$.

DEFINITION 28 (Translation for Method Decomposition). Let $h$ be a compound task that needs to be accomplished, and let $m_i$ be a method that is applicable to $h$. Then, the decomposition of $h$ by $m_i$ is encoded by the following rules:

$$\begin{array}{lll}
\mathit{taskTBA}(t_1, T) & :- & \mathit{method}_i(h, T), \\
 & & \bigwedge_{p \in \rho} \mathit{checked\_state}(p, \chi_1, \chi_2, \ldots, \chi_f, T). \\
\mathit{taskTBA}(t_2, T_2) & :- & \mathit{method}_i(h, T), \\
 & & \bigwedge_{p \in \rho} \mathit{checked\_state}(p, \chi_1, \chi_2, \ldots, \chi_f, T), \\
 & & CAUSABLE(t_1, T, T_2), \\
 & & T_2 \geq T. \\
\vdots & \vdots & \vdots \\
\mathit{taskTBA}(t_n, T_n) & :- & \mathit{method}_i(h, T), \\
 & & \bigwedge_{p \in \rho} \mathit{checked\_state}(p, \chi_1, \chi_2, \ldots, \chi_f, T), \\
 & & CAUSABLE(t_{n-1}, T_{n-1}, T_n), \\
 & & T_n \geq T_{n-1}.
\end{array}$$

where $\chi_1, \ldots, \chi_f$ are the unbound variables of the precondition $p$, and $t_1, \ldots, t_n$ are the subtasks of $m$ –i.e., the ordered list of tasks in the decomposition list of $m$.

The translation in Definition 28 encodes the fact that the decomposition of each subtask $t_k$ can only be done if the previous subtask $t_{k-1}$ has been already accomplished – i.e. $t_{k-1}$ has been already $CAUSABLE$. The only exception is the first task, which is decomposed *if and only if* the particular method, $h$, has been chosen to be applied to the current task in the planning process. This property is encoded by using the $CAUSABLE(t_k, T_k, T_{k+1})$ construct for each subtask $t_k$ (see Definition 18). The time point $T_k$ denotes the time when the current task is decomposed and $T_{k+1}$ denotes the time when it is accomplished – i.e. *causable*. By this way, we can identify the exact place where a specific task is causable in the entire search tree.

Finally, we have the definition of the accomplishment of a task by a particular method.

DEFINITION 29 (Translation for Accomplishment of Compound Tasks). Let $h$ be a compound task that needs to be accomplished, let $m_i$ be a method that is applicable to $h$, and suppose that $h$ has been already decomposed into its subtasks $t_k$ specified by the decomposition list of $m_i$. Then, the accomplishment (i.e., causation) of $h$ by the method $m_i$ is encoded as follows:

$$\begin{array}{lll}
\mathit{causable}(h, T, T_{n+1}) & :- & \mathit{method}_i(h, T), \\
 & & \bigwedge_{p \in \rho} \mathit{checked\_state}(p, \chi_1, \chi_2, \ldots, \chi_f, T), \\
 & & CAUSABLE(t_n, T_n, T_{n+1}), \\
 & & T_{n+1} \geq T_n.
\end{array}$$

where $\chi_1, \ldots, \chi_f$ are the unbound variables of the precondition $p$ of $m$.

Intuitively, $h$ is accomplished (i.e., caused) when/if the last subtask in its decomposition by $m_i$ is accomplished (i.e., caused.) This is a direct consequence of Theorem 14.

Note that some of the rules given Definitions 26 and 27 could also be encoded into disjunctive rules, which seems to be conceptually simpler. However, not all ASP systems do handle disjunctions. Therefore, we decided to use non-disjunctive rules. At this point, we refer to the discussion in subsection 5.2.

## 4.3 A Translation Example: An Elevator Domain

One of the planning domains in the AIPS-2000 planning competition was the Miconic-10 Elevator domain. In order to accommodate the representational power of different planning systems, several different versions of this domain were used in the competition. The simplest version, which we will call Miconic-10-simtest,[8] has the following specifications: (1) the planner simply has to generate plans to serve a group of passengers of whom the origin and destination floors are given, and (2) there are no constraints such as satisfying space requirements of passengers or achieving optimal elevator controls.

Below, we describe how to use the techniques in the previous section to translate an HTN version of the domain into an ASP encoding. For the sake of simplicity and clarity, we present our translation methodology on a simplified and modified version of this problem domain in *Smodels* syntax. The *SHOP* axioms, operators, and methods for this problem domain are shown in Figures 3, 4, and 5, respectively. Our complete encodings of the original Miconic-10-simtest domain for both *DLV* and *Smodels* are available at `<http://www.cs.umd.edu/users/ukuter/ASP_Planning/>`.

In our modified elevator example, there is only one person to be transported in a five floor building. The elevator starts its operation at the ground floor. Our passenger is at the top floor and wants to go down to the ground floor. The elevator can move between any two floors in one step; however, this movement can be either slow or fast, depending on the distance between those floors. The fast movement of the elevator depends on the amount of energy available to it. The elevator has initially enough energy for such movements. However, a fast movement decreases the total energy of the elevator by a specific amount. More specifically, a fast movement between two adjacent floors consumes one unit of energy. Unlike fast movements, slow movements do not require energy. The elevator always makes slow movements when it is empty, in order to conserve energy.

[8]We use this name because the domain is available at `<http://www.informatik.uni-freiburg.de/~koehler/elev/simtests.tar.gz>`.

Now, we will describe the basics of the translation process step by step as described in the previous section.

**Prelude**

We have to formulate all of the possible atoms that can ever be used during the planning process. Due to the fact that each variable in *Smodels* semantics must have a range of values, we must also define type predicates in our translation as described in the previous section. In a *SHOP* domain description, we do not need to make these definitions about the set of all possible atoms and tasks, nor about the type predicates.

In our elevator example, we have the following rules for specifying the possible atoms:

$$\begin{array}{lll}
atom(boarded(P)) & :- & person(P). \\
atom(goal(P)) & :- & person(P). \\
atom(lift\_at(F)) & :- & floor(F). \\
atom(destination(P,F)) & :- & person(P), floor(F). \\
atom(on\_floor(P,F)) & :- & person(P), floor(F). \\
atom(on\_lift(P)) & :- & person(P). \\
atom(total\_energy(E)) & :- & energy\_levels(E).
\end{array}$$

And also the following rules for the type predicates such as:

$$\begin{array}{ll}
time(0..10) & :- \\
person(p0) & :- \\
floor(0..4) & :- \\
energy\_levels(0..10) & :-
\end{array}$$

Note that we define the time line of our logic program to be the set $\{0, 1, \ldots, 10\}$. This definition is just for illustrative purposes in this example. Normally, we use one of the two techniques for defining the time lines, as described in the previous section.

**Encoding the Initial State**

The logic program segment $\mathfrak{Trans}(\mathcal{S})$ consists of the following rules to specify the initial state in our encoding of the elevator example:

$$\begin{array}{ll}
in\_state(lift\_at(0), 0) & :- \\
in\_state(goal(p0), 0) & :- \\
in\_state(on\_floor(p0, 4), 0) & :- \\
in\_state(destination(p0, 0), 0) & :- \\
in\_state(total\_energy(10), 0) & :-
\end{array}$$

As it can be seen from these rules, they specify certain ground atoms to be in the initial state of the planner (Definition 16 in the previous section).

```
;; SHOP Axioms for the simplified Miconic-10-simtest Domain.
 (:- (can-move-fast ?floor1 ?floor2)
       ((> ?floor1 ?floor2)(total_energy ?energy)
       (≥ (- ?floor1 ?floor2) ?energy))
 (:- (can-move-fast ?floor1 ?floor2)
       ((≤ ?floor1 ?floor2)(total_energy ?energy)
       (≥ (- ?floor2 ?floor1) ?energy)))
 (:- (floorDiff ?floor1 ?floor2 ?d)
       ((> ?floor1 ?floor2)(assign ?d (- ?floor1 ?floor2)))
 (:- (floorDiff ?floor1 ?floor2 ?d)
       ((≤ ?floor1 ?floor2)(assign ?d (- ?floor2 ?floor1))))
```

Figure 3. Examples of the *SHOP* axioms for the simplified version of Miconic-10-simtest planning domain.

The last argument for each $in\_state(A, T)$ predicate is the time $T$ at which the atom $A$ holds. As mentioned before, we define the starting time of the planning process to be 0.

The frame axiom $\mathfrak{Trans}(\mathbf{F})$ is as follows:

$$\begin{aligned} in\_state(A, T+1) \quad :- \quad & time(T), atom(A), in\_state(A, T), \\ & not \;\; out\_state(A, T+1). \end{aligned}$$

**Encoding the Goal Task(s).**

Suppose we have a single goal task to accomplish, namely the task of transporting our passenger. Then, the logic program segment $\mathfrak{Trans}(\mathbf{t})$ will encode this task via the following rule:

$$taskTBA(transport\_person, p0, 0) \;\; :-$$

This rule specifies that our goal task to be accomplished at the beginning of planning process is the task ($transport\_person$ $p0$) — i.e., the task of transporting the person $p0$.

**Encoding the Axioms**

Suppose that we have the axioms shown in Figure 3. The intended meaning of these axioms is to decide whether the elevator can make a fast movement between the specified two floors. The criteria for this decision is that if the distance between the two floors is greater than or equal to the total energy of the elevator, then it can move fast between these two floors. The encoding

```
;; SHOP Operators for the simplified Miconic-10-simtest Domain.
 (:operator (!markServed ?person)
        ((goal ?person))
        ())
 (:operator (!moveSlow ?floor1 ?floor2)
        ((lift_at ?floor1))
        ((lift_at ?floor2)))
 (:operator (!moveFast ?floor1 ?floor2 ?old ?new)
        ((lift_at ?floor1)(total_energy ?old))
        ((lift_at ?floor2)(total_energy ?new))
 (:operator (!board ?person ?floor)
        ((on ?person ?floor))
        ((on_lift ?person)))
 (:operator (!debark ?person ?floor)
        ((on_lift ?person))
        ((on ?person ?floor)))
```

Figure 4. Examples of the *SHOP* operators for the simplified version of Miconic-10-simtest planning domain.

of this axiom as the logic program segment $\mathfrak{Trans}(\mathcal{AX})$ is straightforward:

$$\begin{array}{lll}
in\_state(can\_move\_fast(F1, F2), T) & :- & time(T), floor(F1), \\
 & & floor(F2), energy\_level(E), \\
 & & in\_state(total\_energy(E), T), \\
 & & F1 > F2, F1 - F2 \geq E. \\
in\_state(can\_move\_fast(F1, F2), T) & :- & time(T), floor(F1), \\
 & & floor(F2), energy\_level(E), \\
 & & in\_state(total\_energy(E), T), \\
 & & F1 \leq F2, F2 - F1 \geq E. \\
in\_state(floorDiff(F1, F2, F1 - F2), T) & :- & time(T), floor(F1), \\
 & & floor(F2), F1 > F2. \\
in\_state(floorDiff(F1, F2, F2 - F1), T) & :- & time(T), floor(F1), \\
 & & floor(F2), F1 \leq F2.
\end{array}$$

## Encoding the Operators

Suppose that in the domain description of our elevator example, we have the planning operators shown in Figure 4.

The first operator in Figure 4 is for the primitive task of marking a person served. This operator basically removes the $(goal\ ?person)$ atom from the state of the world, which means that the goal of transporting the person $?person$ has been achieved. Note that this operator does not add any atoms to the state of the world. The second operator is for the primitive task of moving the elevator slowly from one floor to another. It simply deletes the $(lift\_at\ ?floor1)$ atom from the state, which describes the location of the elevator before it started its move, and adds the atom $(lift\_at\ ?floor2)$, which describes the location of the elevator after it completed its move. The third operator is for the primitive task of moving the elevator fast. Note that this operator also changes the total amount of energy that the elevator has. The fourth and the fifth operators are for boarding a person to the elevator and for debarking a person from the elevator, respectively.

In our translation, the markServed operator is encoded by a single rule:

$$\begin{array}{lll} out\_state(goal(P), T+1) & :- & time(T), person(P), \\ & & taskTBA\_3(markServed, P, T). \end{array}$$

The encoding of the moveSlow operator corresponds the following:

$$\begin{array}{lll} out\_state(lift\_at(F1), T+1) & :- & time(T), floor(F1), floor(F2), \\ & & taskTBA\_4(moveSlow, F1, F2, T). \\ in\_state(lift\_at(F2), T+1) & :- & time(T), floor(F1), floor(F2), \\ & & taskTBA\_4(moveSlow, F1, F2, T). \end{array}$$

The moveFast, board, and debark operators are encoded similarly.

**Encoding the Methods**

We have the following four methods in our domain description: We have two methods for fast transporting the person from his/her original floor to his/her destination floor. The first method is for the case in which the elevator and the person are on the same floor, so the person can be immediately boarded to the elevator and transported to his/her destination. The second method is for the case in which the elevator and the person are not on the same floor; the elevator must be first moved to the floor of the person so that the person can be transported. Figure 5 shows the *SHOP* specification of these two methods.

We have also two methods for slow transportation of the person, similar to the ones described above. The two groups of methods – i.e., the first two and the second two – correspond to a branching (i.e, backtracking choice) point in the planner's search space, in which different branches may possibly lead to different solution plans. However, the first two methods cannot both yield to a solution due to the way their preconditions are defined. The same is true for the second two as well.

According to Definition 26, the translation of the nondeterministic choice among these four methods is the following set of rules, which correspond to the same branching point in the search space. Here, we only give the rule(s)

```
;; SHOP Methods for the simplified Miconic-10-simtest Domain.
;; Method for FAST transporting a person when the lift is at the same floor
;; as him/her.
 (:method (transport_person ?person)
   ;; preconditions
     ((lift_at ?floor1)(on ?person ?floor1)(destination ?person ?floor2)
       (total_energy ?old)(can_move_fast ?floor1 ?floor2)
       (floorDiff ?floor1 ?floor2 ?d)(assign ?new (- ?old ?d)))
   ;; decomposition task list
     ((!board ?person ?floor1) (!moveFast ?floor1 ?floor2 ?old ?new)
        (!debark ?person ?floor2) (!markServed ?person)))

;; Method for FAST transporting a person when the lift is not at the same floor
;; as the person.
 (:method (transport_person ?person)
   ;; preconditions
     ((lift_at ?floorX)(on ?person ?floor1)(destination ?person ?floor2)
       (total_energy ?old)(can_move_fast ?floor1 ?floor2)
       (floorDiff ?floor1 ?floor2 ?d)(assign ?new (- ?old ?d)))
   ;; decomposition task list
     ((!moveSlow ?floorX ?floor1)(!board ?person ?floor1)
       (!moveFast ?floor1 ?floor2 ?old ?new)(!debark ?person ?floor2)
       (!markServed ?person)))
```

Figure 5. Examples of the *SHOP* methods for the simplified version of Miconic-10-simtest planning domain.

for the first method; the others are almost identical.

$$\begin{array}{l} method\_3\_1(transport\_person, P, T) \ :- \\ \quad time(T), person(P), taskTBA\_3(transport\_person, P, T), \\ \quad not\, method\_3\_2(transport\_person, P, T), \\ \quad not\, method\_3\_3(transport\_person, P, T), \\ \quad not\, method\_3\_4(transport\_person, P, T). \end{array}$$

We now describe the encoding for evaluting the preconditions of this method. For that matter, we need to encode the rules given by Definition 27 for every precondition of this method. As an example, consider the first precondition, which is $(lift\_at\ ?floor1)$ (see Figure 5). This precondition has only one unbound variable; namely, $?floor1$. Then according to Definition 27, we have the following rule:

$$\begin{array}{lll} checked\_state(lift\_at(F1, T) & :- & method\_3\_1(transport\_person, P, T), \\ & & in\_state(lift\_at(F1), T). \end{array}$$

Note that, although $?floor1$ is an unbound variable, we did not create new variable symbols for it and did not use the first rule given in Definition 27. The reason for this is that the elevator can be at one and only one floor at any time in this domain.

The rules for the other preconditions are almost identical to the one above, so we do not give them here due to space limitations (for a complete encoding of this domain, see `<http://www.cs.umd.edu/users/ukuter/ASP_Planning/>`). This finishes our encoding of the precondition evalution for the first method of Figure 5.

The following rules encode the decomposition list of this method:

$$\begin{array}{l} taskTBA\_4(board, P, F1, T)\ :-\\ \quad floor(F1), floor(F2), person(P), energy\_level(Old), number(D),\\ \quad time(T),\ checked\_state(lift\_at(F1), T),\ checked\_state(on(P, F1), T),\\ \quad checked\_state(destination(P, F2), T),\\ \quad checked\_state(total\_energy(Old), T),\\ \quad checked\_state(can\_move\_fast(F1, F2), T),\\ \quad checked\_state(floorDiff(F1, F2, D), T),\\ \quad method\_3\_1(transport\_person, P, T). \end{array}$$

$$\begin{array}{l} taskTBA\_6(moveFast, F1, F2, Old, Old - D, T2)\ :-\\ \quad floor(F1), floor(F2), person(P), energy\_level(Old), number(D),\\ \quad time(T), time(T2), checked\_state(lift\_at(F1), T),\\ \quad checked\_state(on(P, F1), T),\ checked\_state(destination(P, F2), T),\\ \quad checked\_state(total\_energy(Old), T),\\ \quad checked\_state(can\_move\_fast(F1, F2), T),\\ \quad checked\_state(floorDiff(F1, F2, D), T),\\ \quad method\_3\_1(transport\_person, P, T),\\ \quad causable\_5(board, P, F1, T, T2),\ T2 \geq T. \end{array}$$

$$\begin{array}{l} taskTBA\_4(debark, P, F2, T3)\ :-\\ \quad floor(F1), floor(F2), person(P), energy\_level(Old), number(D),\\ \quad time(T), time(T2), time(T3), energy\_level(New),\\ \quad checked\_state(lift\_at(F1), T),\ checked\_state(on(P, F1), T),\\ \quad checked\_state(destination(P, F2), T),\\ \quad checked\_state(total\_energy(Old), T),\\ \quad checked\_state(can\_move\_fast(F1, F2), T),\\ \quad checked\_state(floorDiff(F1, F2, D), T),\\ \quad method\_3\_1(transport\_person, P, T),\\ \quad causable\_7(moveFast, F1, F2, Old, New, T2, T3),\ T3 \geq T2. \end{array}$$

$$\begin{array}{l} taskTBA\_3(markedServed, P, T4)\ :-\\ \quad time(T), time(T3), time(T4), person(P), floor(F1), floor(F2),\\ \quad energy\_level(Old), number(D), method\_3\_1(transport\_person, P, T),\\ \quad checked\_state(lift\_at(F1), T),\ checked\_state(on(P, F1), T),\\ \quad checked\_state(destination(P, F2), T),\\ \quad checked\_state(total\_energy(Old), T),\\ \quad checked\_state(can\_move\_fast(F1, F2), T),\\ \quad checked\_state(floorDiff(F1, F2, D), T),\\ \quad causable\_5(debark, P, F2, T3, T4),\ T4 \geq T3. \end{array}$$

The rules for the decomposition list of the method define the successor subtasks with the order they were specified in that method. Note that in the formalism for HTN planning with Ordered Task Decomposition, the ordering of the subtasks enforces the fact that a subtask $t$ can be selected as the current task for decomposition only if all of the subtasks preceding $t$ are accomplished successfully. This is achieved in our translation by the causable properties of the tasks (see Definition 13).

Note also that although the first method of Figure 5 has an unbound variable ?*new*, we have not encoded this variable as an unbound variable in our rules. This is due to the fact that this variable is used in the method for storing the energy that is left after the elevator moves fast between two levels, and we can encode this as we did in the head of the second rule above. In fact, *SHOP*'s assign statement serves the same purpose; it was a design choice in *SHOP* to handle these cases in the preconditions of a method using an assign statement, rather than handling them in the arguments of the subtasks as we did in our encodings.

At this point, we want to emphasise again that the translation method presented in the previous section is a general technique; one can make several optimisations and modifications during actual implementation.

Now, we are ready to give the rule for accomplishment of the task of transporting the person via the method encoded above:

$$\begin{array}{l} causable\_4(transport\_person, P, T, T5)\ :-\\ \quad time(T), time(T4), time(T5), person(P), floor(F1), floor(F2),\\ \quad energy\_level(Old), number(D), method\_3\_1(transport\_person, P, T),\\ \quad checked\_state(lift\_at(F1), T),\ checked\_state(on(P, F1), T),\\ \quad checked\_state(destination(P, F2), T),\\ \quad checked\_state(total\_energy(Old), T),\\ \quad checked\_state(can\_move\_fast(F1, F2), T),\\ \quad checked\_state(floorDiff(F1, F2, D), T),\\ \quad causable\_4(markedServed, P, T4, T5),\ T5 \geq T4. \end{array}$$

## 5 Results: Theory and Practice

In this section, we present our theoretical results on the correctness and completeness of our translation method. These results in soundness and completeness theorems are for the resulting logic programs under the answer set semantics for planning problems. We also describe in detail the experiments we have undertaken. All the detailed formalisations as well as more implementation related information can be obtained from `<http://www.cs.umd.edu/users/ukuter/ASP_Planning>`.

### 5.1 Soundness and Completeness

Our first theorem states that our translation indeed corresponds to HTN planning as done in *SHOP*. 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 planning problem. A more formal treatment and the fact that *SHOP* is sound and complete is contained in [Nau *et al.*, 2000; 1999].

Let $\mathfrak{Trans}(\cdot)$ be the translation method described in the previous section. Given any HTN planning problem described in *SHOP*'s formalism, we are interested in the relationship between the solutions to the problem and the models (or answer sets) of $\mathfrak{Trans}(\cdot)$.

THEOREM 30 ($\mathfrak{Trans}(\cdot)$ and HTN planning using OTD). *Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is the initial state, $\mathbf{t}$ is the list of ground tasks to be accomplished and $\mathcal{D}$ is the domain description, let $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ be the corresponding logic program with negation. We assume that the set of axioms in $\mathcal{D}$ does not contain any cycles through negation. Furthermore, let $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be the set of solutions as defined in Definition 11. Then,*

1. $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D}) = \emptyset$ *if and only if* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *has no answer sets.*
2. *If* $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D}) \neq \emptyset$*, then the following holds:*
   - *(a) For every plan $P \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$, there is an answer set of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ and a sequence of primitive tasks $t_0, t_1, \ldots, t_n$, such that the predicates $taskTBA(t_i, i)$ that are true in this answer set and the $t_i$ correspond exactly to the steps $p_i$ in $P$.*
   - *(b) For every answer set of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ there is a sequence of primitive tasks $t_0, \ldots, t_n$, such that the predicates $taskTBA(t_i, i)$*

*are true in this answer set, and this sequence constitutes a plan* $[t_0, \dots, t_n] \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$*.*

**Proof.** The proof is in three steps:

> **Step 1:** We take a close look at the *search tree*, introduced in Definition 10, which gives a formal and handy description of the causal theory introduced in Section 4. Through Theorem 14, this search tree is linked with our planning problem as introduced in Definition 8.
>
> **Step 2:** Here we list some facts about the rules used in the translation $\mathfrak{Trans}(\cdot)$. They will be used in the next step.
>
> **Step 3:** We show by induction the precise relationship between branches in the search tree and the existence of stable models and the predicates true in them.

We now give the details of the steps.

**Step 1**

It is immediate that the 2nd entry in the triple $\langle \mathcal{S}', \mathbf{t}_{caused}, \mathbf{t}' \rangle$ represents the caused tasks (as defined in Definition 13): they are all ground and primitive. The operators that are applied to these tasks are the markings on the edges above that node $\langle \mathcal{S}', \mathbf{t}_{caused}, \mathbf{t}' \rangle$. There are also edges marked by methods: they just denote which method has been used in the process to decompose the first compound task in the 3rd entry of the node immediately above it.

As long as the 3rd entry $\mathbf{t}'$ is not yet empty, the tree is expanded until a successful branch is built up. Of course, it can also lead to (1) a dead end: the final node might be unsuccessful, or (2) it might never end and the same methods are applied and lead to longer and longer lists of tasks in $\mathbf{t}'$.

By the very construction of the tree and by the Definition 13, we have the following

$$\begin{gathered}\mathbf{t} \text{ is causable } \mathsf{wrt.}\ (\mathcal{S}, \mathcal{D}) \\ \textit{if and only if} \\ \text{there is a successful branch for the tree for } \langle \mathcal{S}, \emptyset, \mathbf{t} \rangle.\end{gathered}$$

Moreover, comparing the definition of the tree with the original definition of a planning problem, we get that all the plans $\pi \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ are obtained by traversing the successful branches and putting together the edge labellings that correspond to the operator applications (note that these are the ground instances of primitive tasks, i.e., the actions in our plan).

**Step 2**

We now show formally the relation of the tree to the stable models of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$. Several things are worth noticing before we give the formal proof.

1. All the predicates used in $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$ carry as last argument a time variable $T$. This is used, informally, to denote the time when this predicate is active. In a stable model where $taskTBA(t_1,5)$ holds, this is interpreted as *task $t_1$ is to be accomplished at time* 5. Similarly, $method_3(h,7)$ means that the third method for the task $h$ is selected to decompose $h$ at time 7.

2. The first task at time 0 is $h_1$ in $\mathbf{t}'$ (forced by Definition 19) in all stable models of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$.

3. The state predicates represent the current state at any point in time: $in\_state(a,5)$ means that atom $a$ is true at time 5, $out\_state(a,5)$ means that $a$ is not true at time 5. Note that the initial state is encoded in Definition 16, and all changes (when operators are applied) are formalised in Definition 23: $out\_state(.)$ is responsible for the delete lists. The $checked\_state$ predicate uses the state predicates to check whether the preconditions of a method are satisfied in the state at time $T$.

4. The first part of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$ defined in Definition 26 ensures the following: in all stable models of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$, for all time points $T$, exactly one of the methods $m_1,\ldots,m_N$ that are applicable to the current task is selected at time $T$ to decompose it. This is because stable models are minimal.

5. We note that the $taskTBA$ predicate can be true for several tasks at a particular time point. Suppose we have a stable model of $\mathfrak{Trans}((\mathcal{S},\mathbf{t},\mathcal{D}))$ where $taskTBA(t,5)$ holds and $t$ is a compound task. Suppose further that a method $m$ is selected to decompose $t$ (i.e. $method(t,5)$ holds in that stable model), and that the preconditions of $m$ are true in the current state so $t$ can be decomposed into $t_1,t_2$. Then, because of the first rule in Definition 28, $taskTBA(t_1,5)$ is true as well. If $t_1$ were a compound task and a method decomposes it into other tasks, then the first of these tasks would again be the current task at time 5. This goes on as long as a primitive task is found (which would also be true at time 5). In that case, there are no rules of the form in Definition 26 available (since these rules are only stated for compound tasks). Then the time $T$ is incremented by one, due to the $2^{nd}$ case in the definition of $CAUSABLE$.

6. $causable(h, T_1, T_{n+1})$ informally means that the task $h$ is caused at time point $T_{n+1}$ using a method, which was decomposed at time $T_1$. Such a predicate is true in a stable model only if there is a path on the search tree in which $h$ has been decomposed and its subtasks have all been caused.

7. Note that the two rules mentioning the predicate $plan\_found$ (see Step 5. above) ensure that either there is at least one stable model (in which the causable predicate is true) or there are no stable models at all (because there is a negative cycle). This is the only place where there is a potential cycle through negation which could lead to the nonexistence of stable models.

**Step 3**

Let us consider the translation $\mathfrak{Trans}^*((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, which is exactly like $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, except that the clause $plan\_found\ :-not\ plan\_found$ is not included. Note that this is the only cycle through negation in $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ which can be the cause of nonexistence of stable models. The other two places where there are potential conflicts are in $\mathfrak{Trans}(\mathcal{AX})$ and in $\mathfrak{Trans}(\mathcal{METH})$ (via the *checked_state* predicates and the *method* predicates in 1.). However, we explicitly assumed that $\mathcal{AX}$ is free of cycles through negation, and it can be easily seen that the complete instantiation of the *checked_state* predicates is also stratified. Also the rules for the methods ensure that if the $taskTBA$ predicate is true, there are always stable models (there are only even cycles through negation).

It is trivial (but tedious) to show formally by induction on the length of a path in the search tree that the following holds:

1. If *path* is a path in the search tree for $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, then there is an answer set *Ans* of $\mathfrak{Trans}^*((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ such that the following holds:

   (a) For all tasks $t$ and time point $i$: $taskTBA(t, i)$ holds in *Ans* if and only if task $t$ (compound or not) occurs as the first task in $\mathbf{t}'$ at a node of task-depth $i$.

   (b) For all tasks $t$ and time points $i$: $method_j(t, i)$ holds in *Ans* if and only if $method_j$ is the labelling of an edge at task-depth $i$ of the path, for every $j = 1 \ldots N$.

   (c) For all tasks $h$ and time points $i, e, f$: $causable(h, e, f)$ holds in *Ans* if and only if the following holds:

   - the task $h$ has been decomposed at the path at task-depth $e$, and

- all $h$'s successor tasks $t_j$ are causable at time points $g$ such that $e \leq g \leq f$ (i.e. $causable(t_j, \cdot, g_j)$ holds in $Ans$ for all successor tasks $t_j$ of $h$, where we have $e \leq g_j \leq f$).

(d) For all atoms $a$ and time point $i$: $in\_state(a, i)$ holds in $Ans$ if and only if $a$ is true in the state represented by the node of task-depth $i$ in the path.

2. If $Ans$ is an answer set of $\mathfrak{Trans}^*((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, then there is a path in the search tree for $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ such that the above properties 1(a)–1(d) hold.

Note that the above formulation includes the situation where the planning problem has no solution. In that case, there is no successful path in the search tree. But there still might be infinite paths. These are generated because tasks are decomposed without being replaced, eventually, by primitive tasks. So even for those paths, there are corresponding stable models (in which the time variable $T$ is unbounded). Of course, the predicate $plan\_found$ does not hold in these stable models. Thus, if we include the rule $plan\_found : -\, not\ \ plan\_found$, then we get the desired result: successful branches exist if and only if there exist stable models. ■

COROLLARY 31 (Soundness and Completeness of $\mathfrak{Trans}(\cdot)$). *The answer sets of* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *correspond exactly to the plans in* **Sol** *($\mathcal{S}$,$\mathbf{t}$,$\mathcal{D}$). There is a bijection between these two sets and each plan in* **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.*

COROLLARY 32 (Soundness and Completeness of $\mathfrak{Trans}(\cdot)$ wrt *SHOP*). *If the axioms $\mathcal{AX}$ in $\mathcal{D}$ do not contain any (positive or negative) cycles, then the answer sets of* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *correspond exactly to the plans computed by SHOP.*

The corollary follows easily from the theorem and the fact that *SHOP* itself has been shown to be a sound and complete planner.

Note that the only reason why we assume the axiomatic theory in $\mathcal{D}$ is free of cycles is because the *SHOP* implementation cannot handle such axioms. In principle, such axioms would cause no problem for the abstract SHOP procedure, which makes no commitment about what kind of inference procedure to use. The soundness and completeness result for the abstract *SHOP* procedure [Nau *et al.*, 1999; 2000] says something like "if the inference procedure is sound and complete, then so is the planning procedure." However, in the inference procedure used in the implementation of *SHOP*, *SHOP* would go into an infinite loop even for simple axioms like $a \leftarrow a$.

So our overall result (all plans returned by *SHOP* are also obtained using our ASP framework) still holds, even without this assumption. In fact, our method computes plans that an ideal version of *SHOP* would compute as well. It is thus complete for such a version of *SHOP*.

## 5.2 Experimental Study

In our experiments, we used the following three different planning domains:

**The Simple-Travel Domain** This domain is one of the domains included in the distribution of the *SHOP* planning system. The scenario for the domain, as described in [Nau *et al.*, 1999; 2000], is that we want to travel from one location to another in a city. We have three locations: downtown, uptown, and park. There are two possible means of transportation: by taxi and by bus. Taxi travel involves hailing the taxi, riding to the destination and paying the driver \$1.50 plus \$1.00 for each mile travelled. Bus travel involves hailing the bus, paying the driver \$1.00, and riding to the destination. Thus, different plans are possible depending on the weather conditions, the distance between our current location and the one we want to go, and how much money we have.

**The Miconic-10-simtest Domain** This is the domain as described in Section 4.3. It is contained in a series of benchmarks `<http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html>` and it was recently used not only to measure the performance of various planners but also for other translation methods from planning problems into ASP (see `http://www.fcs.nmsu.edu/~tson/asp_planner/>`.

**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).[9] This domain involves transporting people around in planes, using different modes of movement: fast and slow. There were four versions of this domain in the competition; namely STRIPS, NUMERIC, SIMPLE-TIME, and TIME. In the NUMERIC version, aircrafts consume fuel at different rates according to the mode of travel. Using a small set of symbolic fuel levels, the SIMPLE-TIME version manages to combine the benefits of fast travel (shorter journey times) with the associated costs (higher fuel consumption) that must be balanced with the cost of refuelling to arrive at time-efficient plans. In the TIME version, the domain uses

[9]IPC-2002 was organised within the Sixth International Conference on AI Planning and Scheduling 2002 (AIPS-2002). For more information on AIPS-2002 and IPC-2002, please see `<http://www.dur.ac.uk/d.p.long/competition.html>`.

Table 1. Comparison between our *Smodels* encoding of Miconic-10-simtest and the encoding described in [Son *et al.*, 2001]. In this table, "no solution" means that the computer used in these experiments ran out of memory.

| **Problem** | $\mathfrak{Trans}(\cdot)$ | **[Son *et al.*, 2001]** |
|---|---:|---:|
| S1-0 | 0.160 | 8.29 |
| S2-0 | 1.160 | 73.41 |
| S3-0 | 4.450 | 162.24 |
| S4-0 | 12.790 | 964.01 |
| S5-0s1 | 44.090 | no solution |
| S5-0s2 | 44.490 | no solution |
| S6-0 | 46.300 | no solution |

numbers to encode fuel consumptions, dependent on distances, and speeds to calculate travel times in each travel mode. This version is essentially the domain used to illustrate the Zeno planning system developed by Penberthy and Weld [`<http://www.cs.washington.edu/ai/zeno.html>`]. In our experiments we used the STRIPS version of the domain.

We describe our experiments in the following subsections. In these experiments, we used the *Smodels* system v2.27 (which is available at `<http://www.tcs.hut.fi/Software/smodels/>`) and the *DLV* system (available at `<http://www.dbai.tuwien.ac.at/proj/dlv/>`). For the experiments in the *Smodels* system, we used *lparse* v1.0.11 as a grounding front-end. We ran our experiments on an HP Notebook PC with an AMD 900Mhz Processor and 256MB RAM running Linux RedHat v7.2 operating system. In all of our experiments, we were finding all of solutions to each planning problem.

Note that we also redid the experiments of [Son *et al.*, 2001] on our machine so that fair comparisons could be done. All of our source codes are available at `<http://www.cs.umd.edu/users/ukuter/ASP_Planning/>`. In our experiments on the Simple-Travel Domain using our method together with *DLV*, we got a speed-up of two orders of magnitude compared to *Smodels*.

**Comparing our method with [Son *et al.*, 2001]**

This section describes our comparison of the time performance of the logic programs produced by using our translation methodology with that of the logic-program encodings presented in [Son *et al.*, 2001].

Note that the encodings proposed in [Son *et al.*, 2001] do not produce actual HTN encodings. Instead, they make use of only a few properties of HTNs, in order to implement some control knowledge in logic programs that perform classical . In their paper, Son *et al.* showed that employing that control knowledge has improved the time performance of the logic program that encodes an action-based planner.

For our experimental comparison, we had to be very careful about how we wrote our HTN formulation of the planning domain. If two different formulations of a planning problem perform differently, then there are several different ways in which this can occur:

- The two formulations may be based on different ways of conceptualising the problem. For example, one formulation might involve reasoning about the movement of the elevator and where it needs to go next, and another formulation might involve reasoning about the movement of the people and where they need to go next. These two problem formulations would produce very different search spaces.
- The two different formulations may use basically the same tasks, and use them to mean basically the same thing. However, one formulation may take less time because it has lower overhead, or because it does a better job of deciding which tasks should actually be generated and explored.

For our experiments, we did not want to use a different conceptual representation than the one used by [Son *et al.*, 2001], because we wanted our experiments to test the performance of the two different approaches, not our ability to devise a clever conceptual representation! Thus, we were careful to write our HTN formulation of Miconic-10-simtest so that we used basically the same conceptual representation that they did.

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 [Son *et al.*, 2001], which were also obtained on the *Smodels* system.

In our experiments, the logic programs produced by our translation methodology were about 1.5 to 2 orders of magnitude faster than the logic programs produced by the methodology described in [Son *et al.*, 2001]. In addition, our encoding was able to solve problems for which the encoding in [Son *et al.*, 2001] ran out of memory. In this respect, these results confirm the fact that a *SHOP*-like HTN planning approach is a much more effective way for solving planning problems if a good set of HTN methods is available, because the HTN methods constrain the size of the search space. They also illustrate that our translation method provides a way to produce efficient HTN-logic programs with ASP semantics to solve planning problems compared to action-based encoding methodologies in the style of [Son *et al.*, 2001].

Table 2. Comparing *Smodels* and *DLV* on the Simple-Travel Domain

| **Problem** | *Smodels* | *DLV* | *DLV* grounding+*Smodels* |
|---|---|---|---|
| P1 | 3.430 | 0.050 | 0.040+0.020 |
| P2 | 3.330 | 0.050 | 0.050+0.020 |
| P3 | 3.190 | 0.030 | 0.030+0.000 |
| P4 | 3.340 | 0.070 | 0.060+0.010 |
| P5 | 3.410 | 0.060 | 0.050+0.030 |
| P6 | 3.230 | 0.030 | 0.030+0.010 |
| P7 | 3.340 | 0.050 | 0.050+0.010 |
| P8 | 3.260 | 0.040 | 0.040+0.010 |
| P9 | 3.230 | 0.040 | 0.040+0.010 |
| P10 | 3.410 | 0.070 | 0.050+0.000 |
| P11 | 3.340 | 0.050 | 0.040+0.000 |
| P12 | 3.250 | 0.020 | 0.030+0.010 |
| P13 | 3.410 | 0.070 | 0.060+0.010 |
| P14 | 3.350 | 0.060 | 0.050+0.020 |
| P15 | 3.270 | 0.030 | 0.030+0.000 |
| P16 | 3.380 | 0.060 | 0.050+0.010 |
| P17 | 3.300 | 0.050 | 0.050+0.010 |
| P18 | 3.260 | 0.030 | 0.030+0.000 |

**Comparing *Smodels* and *DLV* using planning benchmarks**

We believe 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 unbound variables in the programs, or, at least, produces an intelligent grounding. Otherwise, the system tries to make every rule ground in the input program, which decreases the efficiency of planning by causing a combinatorial explosion in the size of the search space.

As we described earlier, neither *Smodels* nor *DLV* is designed to work on the logic programs with unbound variables. However, *DLV*'s grounding differs from that of *Smodels*, and is generally believed to be *more intelligent* and smaller in size. Also, *DLV* is based on deductive database techniques and we expected a better handling of unbound variables. To test this hypothesis, we applied our translation methodology to our elevator and travelling examples. While testing this hypothesis, one must note the way the *Smodels* system solves a problem. *Smodels* uses a front-end called *lparse* to preprocess the programs in order to get them grounded. After the programs are grounded, the *Smodels* ASP solver solves the ground program itself. Therefore, the problems solved in this system may not reflect the ac-

Table 3. Comparison of *Smodels* and *DLV* using $\mathfrak{Trans}(\cdot)$ on the Miconic-10-simtest Domain

| **Problem** | *Smodels* | *DLV* | *DLV* grounding+*Smodels* |
|---|---|---|---|
| S1-0 | 0.160 | 0.040 | 0.030+0.010 |
| S2-0 | 1.160 | 0.060 | 0.050+0.010 |
| S3-0 | 4.450 | 0.080 | 0.010+0.090 |
| S4-0 | 12.790 | 0.260 | 0.100+0.530 |
| S5-0s1 | 44.090 | 0.640 | 0.080+1.540 |
| S5-0s2 | 44.490 | 0.680 | 0.090+1.840 |
| S6-0 | 46.300 | 0.980 | 0.170+3.560 |

tual performance of the ASP solver. To accommodate this issue, we also designed a set of experiments in which we used *DLV* to produce groundings of the programs. These groundings were then given to *Smodels*. The aim in these experiments was to determine the effect of grounding done by the front-end *lparse* on the overall performance of the *Smodels* system.

Tables 2 and 3 show our results on the Simple-Travel and Miconic-10-simtest problems. We compared our encodings using *Smodels* with *lparse* for grounding, *DLV*, and *Smodels* with *DLV* for grounding. These results suggest that our programs are much faster on *DLV* than on *Smodels* using *lparse* for grounding. One possible reason for this behavior is as follows. In *Smodels* we have to define type predicates for each variable in the problem domain description as well as all possible ground instances of the atoms that can ever be used in the planning process. The result is that as the number of variables and the number of their possible instantiations increase, the time performance of the logic program decreases. However, we do not have such constraints in *DLV*.

In order to test the effect of grounding in two systems, we have also designed the following experiments. We used *DLV* to produce the ground programs, and fed these into *Smodels*, instead of using *lparse* for grounding. The result is that the overall performance of *Smodels* is increased to almost that of *DLV* on the Simple-Travel problems. Note that the last column in Table 2 contains the sum of (1) the time for producing the grounded version by *DLV*, and (2) the time it takes for *Smodels* to produce the solution based on this grounding.

We observed similar behaviour between the two systems for our elevator problems. As can be seen in Table 3, *DLV* performed better than *Smodels* not only in direct comparison, but also when *Smodels* used the grounding obtained by *DLV*. The reasons behind the differences in the performance of both *DLV* and *Smodels* on the problems in Miconic-10-simtest and Simple-

Travel domains are twofold: (1) the number of solutions for a problem, and (2) the task depth – i.e., the length of a solution. In Simple-Travel domain, the hardest problems has only 3 solutions, whereas the hardest problem in Miconic-10-simtest domain has 120 solutions. Furthermore, the length of the solutions in Simple-Travel domain is at 4 — i.e., there are 4 steps in the longest plan found in this domain. However, in Miconic-10-simtest domain, we have solutions with 20 actions in them. This means that the search trees that correspond to Miconic-10-simtest problems are much larger than the ones that correspond to Simple-Travel problems. As a result, our programs required much more time to find all of the solutions for the problems in Miconic-10-simtest domain than they required for Simpe-Travel problems.

Note that, in Miconic-10-simtest problems, the performance of *Smodels* using *DLV*'s groundings is not as good as that of *DLV* on the same problems. One possible explanation for *DLV*'s dominance in performance over *Smodels* on these problems is that the implemented search heuristics for the *guess-and-check* method in *DLV* are better suited for planning problem such as the ones in these experiments.

The above results led us to investigate more about the performances of the ASP solvers (without the grounding part) of the two systems. To do this, we designed a new set of experiments using the Zeno-Travel domain, which was introduced as a benchmark problem in the recent AIPS-2002 planning competition. These experiments involved harder problems than the previous ones: the hardest one (e.g. p25 in Table 4) has over 20000 solutions (i.e stable models), whereas no problem in Tables 2 and 3 has more than 120 models. In these experiments, we compared the performance of *Smodels* using *lparse* for grounding with that of *Smodels* using ground programs produced by *DLV*. Therefore, we were able to investigate the performances of the ASP solvers implemented in the two systems.

The results for these experiments are shown in Table 4. *Smodels* using *lparse* for grounding was only able to solve the first two problems, and on those problems its performance was about an order of magnitude worse than that of *DLV*. *Smodels* with *lparse* was unable to solve the rest of the problems because *lparse* ran out of memory. Table 4 also shows the time it takes for *DLV* to ground the problems. As it can be seen, *DLV* performs better than *Smodels*, even when *Smodels* is using the grounding produced by *DLV*. These results clearly indicate the difference in model generation algorithms implemented in the two systems. According to these results, we can conclude that *DLV* is better suited for the encodings of planning with ordered task decomposition.

Note that on the hardest problems, namely P22-P25, of Zeno Travel, *Smodels* could not find a solution (Table 4). This was not because the

Table 4. Comparison of *Smodels* and *DLV* on the Zeno-Travel Domain. In this table, "no solution" means that the computer used in these experiments ran out of memory.

| **Problem** | *DLV* | *DLV* grounding+*Smodels* |
|---|---|---|
| P1 | 0.590 | 0.510+0.330 |
| P2 | 0.670 | 0.590+0.330 |
| P3 | 0.410 | 0.380+0.060 |
| P4 | 0.320 | 0.290+0.040 |
| P5 | 0.490 | 0.490+0.080 |
| P6 | 0.360 | 0.350+0.040 |
| P7 | 16.440 | 14.340+35.210 |
| P8 | 26.180 | 22.630+85.390 |
| P9 | 38.390 | 36.160+76.860 |
| P10 | 27.220 | 24.840+52.730 |
| P11 | 30.370 | 28.150+55.550 |
| P12 | 22.930 | 20.930+21.310 |
| P13 | 16.560 | 14.920+22.650 |
| P14 | 18.230 | 16.240+66.310 |
| P15 | 17.020 | 14.960+38.190 |
| P16 | 78.060 | 70.880+152.190 |
| P17 | 66.300 | 62.450+62.75 |
| P18 | 85.000 | 81.370+194.940 |
| P19 | 146.030 | 139.700+138.240 |
| P20 | 168.630 | 163.660+329.940 |
| P21 | 120.080 | 117.160+106.330 |
| P22 | 2025.16 | 1578.69+no solution |
| P23 | 4275.25 | 4236.60+no solution |
| P24 | 3612.96 | 3462.32+no solution |
| P25 | 4619.24 | 4585.35+no solution |

ASP solver itself could not solve the problems, but the ground programs generated by *DLV* were too big for *Smodels*'s front-end *lparse* to parse and convert them into the input syntax required by *Smodels*. On these problems, *lparse* ran out of memory and the operating system killed its process.

**Influence of using disjunctions**

Finally, we did another set of experiments to find out the influence of using disjunctions in our transformation. In Definition 26 we used a set of rules to represent nondeterministic choice of methods. Conceptually, these rules

Table 5. Comparison of *Smodels* and *DLV* on the Simple-Travel Domain using disjunctions

| **Problem** | *Smodels* | *DLV* | *DLV* grounding+*Smodels* |
|---|---|---|---|
| P1 | 3.640 | 0.050 | 0.050+0.010 |
| P2 | 3.560 | 0.050 | 0.030+0.000 |
| P3 | 3.360 | 0.030 | 0.030+0.000 |
| P4 | 3.590 | 0.080 | 0.060+0.010 |
| P5 | 3.610 | 0.050 | 0.050+0.000 |
| P6 | 3.500 | 0.030 | 0.030+0.000 |
| P7 | 3.580 | 0.050 | 0.050+0.020 |
| P8 | 3.480 | 0.040 | 0.030+0.010 |
| P9 | 3.430 | 0.030 | 0.030+0.000 |
| P10 | 3.560 | 0.050 | 0.050+0.010 |
| P11 | 3.550 | 0.040 | 0.030+0.010 |
| P12 | 3.450 | 0.020 | 0.010+0.000 |
| P13 | 3.570 | 0.060 | 0.050+0.020 |
| P14 | 3.550 | 0.050 | 0.050+0.020 |
| P15 | 3.470 | 0.030 | 0.030+0.000 |
| P16 | 3.580 | 0.050 | 0.050+0.010 |
| P17 | 3.460 | 0.040 | 0.040+0.010 |
| P18 | 3.510 | 0.030 | 0.030+0.000 |

can be much more easily formulated by just one disjunctive rule:

$$\begin{aligned} &\mathit{method}_1(h,T) \vee \mathit{method}_2(h,T) \vee \ldots \vee \mathit{method}_N(h,T) \\ &\qquad \leftarrow \ \mathit{taskTBA}(h,T). \end{aligned}$$

Tables 5 and 6 show the results. While disjunctions pay off for the simpler problem (times are slightly better), they do not pay off for the harder Zeno-Travel problems. We do not have an explanation for this yet. We had expected that since *DLV* is especially designed to deal with disjunctions, it would perform better when using disjunctions than when we coded them into equivalent non-disjunctive versions.

## Comparison with SHOP

Encouraged by the performances of the logic programs produced by our translation, we prepared a set of experiments to compare the time performances of our logic-program encodings on both the Simple-Travel and Zeno-Travel examples with those of the SHOP planning system itself.

Tables 7 and 8 augment our results from Tables 2 and 3 to include comparisons with *SHOP* on the Simple-Travel and Miconic-10-simtest domains.

Table 6. Comparison of *Smodels* and *DLV* on the Zeno-Travel Domain with disjunctions

| **Problem** | *DLV* | *DLV* grounding+*Smodels* |
|---|---|---|
| P1 | 0.600 | 0.510+0.340 |
| P2 | 0.650 | 0.620+0.310 |
| P3 | 0.410 | 0.410+0.070 |
| P4 | 0.330 | 0.310+0.040 |
| P5 | 0.520 | 0.490+0.080 |
| P6 | 0.380 | 0.370+0.050 |
| P7 | 19.390 | 14.420+34.790 |
| P8 | 29.990 | 22.720+84.950 |
| P9 | 43.150 | 35.630+75.690 |
| P10 | 30.960 | 24.630+53.670 |
| P11 | 34.170 | 27.920+56.080 |
| P12 | 25.220 | 20.890+20.590 |
| P13 | 18.530 | 14.620+22.550 |
| P14 | 20.540 | 16.060+70.750 |
| P15 | 19.460 | 14.880+39.430 |
| P16 | 89.050 | 70.640+150.000 |
| P17 | 73.030 | 62.540+63.600 |
| P18 | 90.510 | 81.300+194.900 |
| P19 | 162.110 | 142.210+133.450 |
| P20 | 171.910 | 164.510+335.090 |
| P21 | 121.900 | 120.060+106.680 |

For the Simple-Travel domain, the comparison is inconclusive, because the amount of time taken by *SHOP* was too small for us to measure accurately. For the Miconic-10-simtest domain, the time needed by our program using *DLV* was about 1 to 2 orders of magnitude more than the time needed by *SHOP*.

The results of our experiments on the Zeno-Travel Domain can be seen at Table 9. In most cases, the time needed by our program using *DLV* was 2.5 to 3.5 orders of magnitude more than *SHOP*.

The experimental results are encouraging in several ways:

- To the best of our knowledge, this is the first time that an ASP-based approach has been able to do this well on planning problems of this calibre. Our ASP programs were slower than *SHOP*, but this is to be expected since *SHOP*, and also its successor *SHOP2*, are known to be efficient planning systems. For example, *SHOP2* was one of the fastest

Table 7. Comparison of *Smodels* and *DLV* with *SHOP* on the Simple-Travel Domain

| **Problem** | *Smodels* | *DLV* | *DLV* grounding+*Smodels* | *SHOP* |
|---|---|---|---|---|
| P1 | 3.430 | 0.050 | 0.040+0.020 | 0.000 |
| P2 | 3.330 | 0.050 | 0.050+0.020 | 0.000 |
| P3 | 3.190 | 0.030 | 0.030+0.000 | 0.000 |
| P4 | 3.340 | 0.070 | 0.060+0.010 | 0.000 |
| P5 | 3.410 | 0.060 | 0.050+0.030 | 0.000 |
| P6 | 3.230 | 0.030 | 0.030+0.010 | 0.000 |
| P7 | 3.340 | 0.050 | 0.050+0.010 | 0.000 |
| P8 | 3.260 | 0.040 | 0.040+0.010 | 0.000 |
| P9 | 3.230 | 0.040 | 0.040+0.010 | 0.000 |
| P10 | 3.410 | 0.070 | 0.050+0.000 | 0.000 |
| P11 | 3.340 | 0.050 | 0.040+0.000 | 0.000 |
| P12 | 3.250 | 0.020 | 0.030+0.010 | 0.000 |
| P13 | 3.410 | 0.070 | 0.060+0.010 | 0.000 |
| P14 | 3.350 | 0.060 | 0.050+0.020 | 0.000 |
| P15 | 3.270 | 0.030 | 0.030+0.000 | 0.000 |
| P16 | 3.380 | 0.060 | 0.050+0.010 | 0.000 |
| P17 | 3.300 | 0.050 | 0.050+0.010 | 0.000 |
| P18 | 3.260 | 0.030 | 0.030+0.000 | 0.000 |

planning systems in the AIPS-2002 planning competition `http://www.laas.fr/aips/`.

- Consider the "performance ratio" for our programs, i.e., the ratio of the amount of time they require to the time required by *SHOP*. If the average-case time complexity of our programs were worse than that of *SHOP*, then we would expect their performance ratio to get worse with increasing problem size. In our experiments, the performance ratio did not seem to get worse with increasing problem size.
  Although there is not enough data to say so conclusively, this suggests 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*.
- In the HTN formulation of the Miconic-10-simtest domain, the average branching factor (the average number of subtasks of each task) is smaller than in the Zeno Travel domain. The performance ratio for our programs is about two orders of magnitude better in the Miconic-10-simtest do-

Table 8. Comparison of *Smodels* and *DLV* using $\mathfrak{Trans}(\cdot)$ with *SHOP* on the Miconic-10-simtest Domain

| **Problem** | *Smodels* | *DLV* | *DLV* grounding +*Smodels* | *SHOP* | (*DLV* / *SHOP*) Ratio |
|---|---|---|---|---|---|
| S1-0 | 0.160 | 0.040 | 0.030+0.010 | 0.000 | - |
| S2-0 | 1.160 | 0.060 | 0.050+0.010 | 0.010 | 6 |
| S3-0 | 4.450 | 0.080 | 0.010+0.090 | 0.000 | - |
| S4-0 | 12.790 | 0.260 | 0.100+0.530 | 0.020 | 13 |
| S5-0s1 | 44.090 | 0.640 | 0.080+1.540 | 0.060 | 10.67 |
| S5-0s2 | 44.490 | 0.680 | 0.090+1.840 | 0.060 | 11.33 |
| S6-0 | 46.300 | 0.980 | 0.170+3.560 | 0.090 | 10.89 |

main than in the Zeno Travel domain. As explained later in Section 6, a likely explanation is that the ASP systems are creating ground instances of clauses that are irrelevant for the planning process, because the number of such irrelevant ground instances grows combinatorially with the branching factor.

If this hypothesis is correct, then it may be possible to improve the performance of our ASP systems—possibly by several orders of magnitude—if we can avoid creating the ground instances. In the near future, we will test our system on more planning domains and compare our approach with other well-known planning systems. We are also planning to implement our approach on two systems, namely the *XSB* system ([Rao *et al.*, 1997]) and the front-end software developed by P. Bonatti for *Smodels* ([Bonatti, 2001b; 2001a]), both of which can handle unbound variables, unlike *DLV* and *Smodels*.

## 6 Conclusions and Future Research Directions

In this paper, we have described a way to encode HTN planning problems into logic programs under the answer set semantics. This transformation is not only sound and complete, but it also corresponds closely to HTN planning systems which generate plans by using ordered task decompositions. Previous encodings (as first introduced in [Dimopoulos *et al.*, 1997]) either consider only or they take a special view of HTN planning (as constraint-based planning in [Son *et al.*, 2001]).

In general, STRIPS-style preconditions do not have enough expressive power to provide the amount of control that is needed for efficient planning. It is not hard to formulate examples of planning problems for which a STRIPS-style representation of the problem will have a much larger search space than the search space than can be achieved with a more expressive

Table 9. Comparison of *DLV* with *SHOP* on the Zeno-Travel Domain

| **Problem** | *DLV* | *SHOP* | Performance Ratio (*DLV* / *SHOP*) |
|---|---|---|---|
| P1 | 0.590 | 0.000 | - |
| P2 | 0.670 | 0.010 | 67.00 |
| P3 | 0.410 | 0.000 | - |
| P4 | 0.320 | 0.010 | 32.00 |
| P5 | 0.490 | 0.000 | - |
| P6 | 0.360 | 0.000 | - |
| P7 | 16.440 | 0.020 | 822.00 |
| P8 | 26.180 | 0.030 | 872.67 |
| P9 | 38.390 | 0.070 | 548.43 |
| P10 | 27.220 | 0.040 | 680.50 |
| P11 | 30.370 | 0.030 | 1012.34 |
| P12 | 22.930 | 0.020 | 1146.50 |
| P13 | 16.560 | 0.060 | 276.00 |
| P14 | 18.230 | 0.020 | 911.50 |
| P15 | 17.020 | 0.020 | 851.00 |
| P16 | 78.060 | 0.090 | 867.34 |
| P17 | 66.300 | 0.060 | 1105.00 |
| P18 | 85.000 | 0.070 | 1214.29 |
| P19 | 146.030 | 0.120 | 1216.92 |
| P20 | 168.630 | 0.130 | 1297.15 |
| P21 | 120.080 | 0.100 | 1200.80 |
| P22 | 2025.16 | 3.050 | 663.99 |
| P23 | 4275.25 | 12.250 | 349.00 |
| P24 | 3612.96 | 7.980 | 452.75 |
| P25 | 4619.24 | 12.860 | 359.19 |

representation such as *SHOP*'s HTNs, or the temporal-logic control rules used in *TLplan* [Bacchus and Kabanza, 2000] and *TALplanner* [Kvarnström and Doherty, 2001]. The practical result of this is that *SHOP* and *TLplan* and *TALplanner* have been able to solve far more planning problems than STRIPS-style planning systems and have been able to solve them in several orders of magnitude less time in extensive empirical comparisons across dozens of planning domains [Nau *et al.*, 1999; Bacchus and Kabanza, 2000; Bacchus, 2001; Fox and Long, 2002].

To test our approach, we have used it to create both *Smodels* and *DLV* logic programs, for three different AI planning domains: the Miconic-10-simtest domain, the Simple-Travel Domain, and the Zeno-Travel domain.

Here is a summary of our experimental results and what we believe they signify:

**HTN vs. action-based** In our experiments, our *Smodels* logic programs clearly outperformed the corresponding ones described in [Son *et al.*, 2001], which are based on answer set semantics. Our programs took several orders of magnitude less time to solve planning problems, and they solved several problems that were inaccessible to the based ASP systems. This is due largely to the HTN-style control knowledge that our translation methodology encodes into the logic programs. HTN planning is more expressive than [Erol *et al.*, 1996], and in particular, the domain description for an HTN formulation of a domain can include domain-specific knowledge about how to carry out the planning process—knowledge that cannot be expressed in an action-based formalism. To develop this domain-specific knowledge can take a significant amount of human effort, but it can enable the planning system to search a much smaller search space than the search space explored by action-based formalisms.

***DLV* vs. *Smodels*** In our experiments on the Simple-Travel Domain using our method together with *DLV*, we got a speed-up of two orders of magnitude compared to *Smodels*. We believe one of the reasons for this is that *Smodels* requires type predicates to be defined as an input, which creates combinatorially many ground instances of the clauses in the logic program. For any given problem instance, most of these clauses are likely to be irrelevant. But our experiments also revealed that the grounding of *Smodels* is not the only source responsible for this behaviour. By using the grounding obtained from *DLV* and then running *Smodels*, we still got a performance of *Smodels* that does not compete with *DLV*.

***SHOP* vs ASP** In our experiments, our logic-program encodings were not competitive with *SHOP*. That is not particularly surprising, since *SHOP* is a state-of-the-art HTN planning system. However, we find it quite interesting and encouraging that in our experiments, the time requirement for our logic-program encodings did not seem to grow any faster than proportional to *SHOP*'s time requirement.

We believe that most of the difference in performance is due to the grounding mechanism underlying ASP systems. In order to run our logic-program encoding of an HTN domain, both *DLV* and *Smodels* must first ground the program. This can generate many ground instances for the rules that correspond to the methods. For example, suppose that a method $m$ corresponds to $r$ different rules, that the number of unground variables in each rule is $c$, and that the number of possible values for each variable is $k$. Then there will be $rk^c$ ground instances for the rules. Now, suppose we are trying to accomplish a task $t$ that is unified with a method $m$. Then in *DLV* and *Smodels*, the branching factor of the search space may potentially be as high as $rk^c$ in the worst case. At the same node of the search space, the branching factor for *SHOP* will typically be much less, because *SHOP* will be able to use method instances that are only partially ground. Furthermore, if there are $p$ different predicate symbols in the domain description, then there will be $pk^c$ ground instances of the frame axiom for the domain. *SHOP* will not have any of these instances because it does not need the frame axiom.

If the difference in performance is because of the reasons described above, then we should be able to get a great improvement in performance by using systems like XSB. We have just started investigating this aspect.

We emphasise the fact that our method does not use any particular features of the engine for computing answer sets. Obviously, taking advantage of the particular search method of *Smodels*, or the bottom-up evaluation of *DLV*, it would be possible to write even more efficient translations. But our aim is to develop a translation that is independent of the underlying nonmonotonic engine.

As a byproduct, we believe our method can be easily used as a way to transfer benchmarks from the planning community to benchmarks for comparing nonmonotonic systems based on computing answer sets. This is because our method does not rely on the features of a particular ASP system. In the near future, there are several additional investigations that we would like to perform:

- We want to test the benchmarks on the *XSB* system, a Prolog system which not only allows function symbols but also unbound variables at the same time. These are features that neither *Smodels* nor *DLV* provide. We believe that we can get a competitive planning system once we can apply our translation into a nonmonotonic system with these two features.
- In all of our experiments, we were finding all of solutions to each planning problem. However, in most planning domains, one just wants to find a single solution (hopefully a near-optimal one). In *SHOP*, finding a single solution takes exponentially less time than finding all solutions. Whether it takes exponentially less time for our logic-program encodings is an open question, and we would like to run additional experiments to find out.

- We are also planning to compare our method with *Smodels* equipped with a front-end to allow for (restricted use of) unbound variables ([Bonatti, 2001b; 2001a]). The latter system has been developed by Piero Bonatti and is a front-end system that can be added to any system computing answer sets and based on grounding. This would also allow for comparisons of systems with built-in grounding to those who do not require this (but are, in general, slower). Again, we believe that serious benchmarks from the planning community can help a lot to evaluate nonmonotonic systems.

Our overall aim is 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 can learn from the different translation methods. We expect that optimal translations (if they exist) depend on the particular application area. Developing a methodology to determine or classify such domains seems to us to be worthwhile.

## 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. Opinions expressed in this paper are those of authors and do not necessarily reflect opinion of the funders.

## BIBLIOGRAPHY

[Aarup *et al.*, 1994] M. Aarup, M. M. Arentoft, Y. Parrod, J. Stader, and I. Stokes. OPTIMUM-AIV: A Knowledge-based Planning and Scheduling System for Spacecraft AIV. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, pages 451–469. Morgan Kaufmann, 1994.

[Agosta, 1995] J. M. Agosta. Formulation and Implementation of an Equipment Configuration Problem with the SIPE-2 Generative Planner. In *Proceedings of AAAI-95 Spring Symposium on Integrated Planning Applications*, pages 1–10, 1995.

[Apt *et al.*, 1999] K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors. *The Logic Programming Paradigm: Current Trends and Future Directions*, Berlin, 1999. Springer.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[Bacchus, 2001] F. Bacchus. AIPS'00 Planning Competition. *AI Magazine*, 22(3):47–56, 2001.

[Baral *et al.*, 2002] C. Baral, N. Tran, and L. Tuan. Reasoning about Actions in a Probabilistic Setting. In *AAAI/IAAI 2002*, pages 507–512. AAAI Press, 2002.

[Biundo and Schattenberg, 2001] S. Biundo and B. Schattenberg. From Abstract Crisis to Concrete Relief. A Preliminary Report on Flexible Integration on Nonlinear and Hierarchical Planning. In A. Cesta and D. Borrajo, editors, *Proceedings of the 6th European Conference on Planning (ECP-01)*, LNAI, pages 157–168, Toledo, Spain, 2001. Springer-Verlag.

[Bonatti, 2001a] P.A. Bonatti. Prototypes for Reasoning with Infinite Stable Models and Function Symbols. In Th. Eiter, M. Truszczyński, and W. Faber, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, pages 416–419, Berlin, September 2001. Springer.

[Bonatti, 2001b] P.A. Bonatti. Reasoning with Infinite Stable Models. In B. Nebel, editor, *Proceedings of IJCAI-01*, pages 603–608, Seattle, Washington, August 2001. Morgan Kaufmann.

[Chen and Warren, 1996] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[Dimopoulos *et al.*, 1997] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Non-monotonic Logic Programs. In S. Steel and R. Alami, editors, *Proceedings of the Fourth European Conference on Planning*, pages 169–181, Toulouse, France, September 1997. Springer-Verlag.

[Dix *et al.*, 2001] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In Andrei Voronkov and Alan Robinson, editors, *Handbook of Automated Reasoning*, pages 1121–1234. Elsevier-Science-Press, 2001.

[Dix *et al.*, 2002] Jürgen Dix, Hector Munoz-Avila, and Dana Nau an Lingling Zhang. Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of Journees Europeens de la Logique en Intelligence Artificielle (JELIA '02)*, LNCS 2424, pages 173–185. Springer, 2002.

[Dix *et al.*, 2003] Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang. IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI*, 37(4):381–407, 2003.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann, 1998.

[Eiter *et al.*, 2002] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. The DLV-K Planning System: Progress Report. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of Journees Europeens de la Logique en Intelligence Artificielle (JELIA '02)*, LNCS 2424, pages 541–544. Springer, 2002.

[Eiter *et al.*, 2003] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Planning Under Action Costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2003.

[Erol *et al.*, 1994] K. Erol, J. Hendler, and D.S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In K. Hammond, editor, *Proceedings of AIPS-94*, pages 249–254, Chicago, IL, 1994. AAAI Press.

[Erol *et al.*, 1996] K. Erol, J. Hendler, and D. Nau. Complexity Results for Hierarchical Task-Network Planning. *Annals of Mathematics and Artificial Intelligence*, 18:69–93, 1996.

[Estlin *et al.*, 1997] T. A. Estlin, S. A. Chien, and X. Wang. An Argument for Hybrid HTN/Operator-Based Approach to Planning. In S. Steel and R. Alami, editors, *Proc. Fourth European Conference on Planning (ECP-97)*, pages 184–196, Toulouse, France, September 1997. Springer-Verlag.

[Fox and Long, 2002] M. Fox and D. Long. International planning competition. `http://www.dur.ac.uk/d.p.long/competition.html`, 2002.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[Gelfond and Lifschitz, 1998] M. Gelfond and V. Lifschitz. Action Languages. *Electronic Transactions on AI*, 2(3–4):193–210, 1998.

[Giunchiglia and Lifschitz, 1998] E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proc. AAAI-98*, pages 623–630, Madison, Wisconsin, 1998. AAAI Press.

[Hebbar *et al.*, 1996] K. Hebbar, S.J. Smith, I. Minis, and D. Nau. Plan-Based Evaluation of Designs for Microwave Modules. In *Proc. ASME Design Technical Conference and Computers in Engineering Conference*, Irving, California, August 1996.

[Kvarnström and Doherty, 2001] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Articial Intelligence*, 30:119–169, 2001.

[Lifschitz, 1999] V. Lifschitz. Action Languages, Answer Sets and Planning. In K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 357–373. Springer-Verlag, 1999.

[Lifschitz, 2002] V. Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138(1–2), 2002.

[Marek and Truszczyński, 1999] W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Apt et al. [1999], pages 375–398.

[McCain and Turner, 1997] N. McCain and H. Turner. Causal Theories of Action and Change. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 460–465, Menlo Park, CA, 1997. AAAI Press.

[McCain, 1999] N. McCain. Using Causal Calculator with the C Input Language. Technical report, University of Texas at Austin, 1999.

[Muñoz-Avila *et al.*, 2001] H. Muñoz-Avila, D.W. Aha, D.S. Nau, R. Weber, L. Breslow, and F. Yaman. SiN: Integrating Case-based Reasoning with Task Decomposition. In B. Nebel, editor, *Proceedings of IJCAI-2001*, Seattle, Washington, August 2001. Morgan Kaufmann.

[N. Leone and Scarcello, 2001] S. Perri N. Leone and F. Scarcello. Improving ASP instantiators by join-ordering methods. In Th. Eiter, M. Truszczyński, and W. Faber, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, pages 280–294, Berlin, September 2001. Springer.

[Nau *et al.*, 1999] D.S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In T. Dean, editor, *Proceedings of IJCAI-99*, pages 968–973. Morgan Kaufmann, 1999.

[Nau *et al.*, 2000] D.S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP and M-SHOP: Planning with Ordered Task Decomposition. Technical Report CS TR 4157, University of Maryland, 2000.

[Nau *et al.*, 2001] D.S. Nau, Y. Cao, A. Lotem, H. Muñoz-Avila, and S. Mitchell. Total-Order Planning with Partially Ordered Subtasks. In B. Nebel, editor, *Proceedings of IJCAI-01*, pages 425–430, Seattle, Washington, August 2001. Morgan Kaufmann.

[Nau *et al.*, 2003] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, December 2003.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[Niemelä, 1999] Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. volume 25(3–4) of *Special Issue of the Annals in Mathematics and Artificial Intelligence*, pages 241–273. Baltzer Science Publishers, 1999.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, 1988.

[Rao *et al.*, 1997] Prasad Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference*, LNAI 1265, pages 430–440, Berlin, June 1997. Springer.

[Sacerdoti, 1990] Earl D. Sacerdoti. The Nonlinear Nature of Plans. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 162–170. Morgan Kaufmann, 1990. Originally appeared in *Proc. IJCAI-75*, pp. 206-214.

[Smith *et al.*, 1997] S. J. Smith, K. Hebbar, D. Nau, and I. Minis. Integrating Electrical and Mechanical Design and Process Planning. In M. Mantyla, S. Finger, and T. Tomiyama, editors, *Knowledge Intensive CAD*, volume 2, pages 269–288. Kluwer Academic Publishers, Boston, 1997.

[Smith *et al.*, 1998a] S. J. Smith, D. S. Nau, and T. Throop. Success in Spades: Using AI Planning Techniques to Win the World Championship of Computer Bridge. In *AAAI-98/IAAI-98*, pages 1079–1086. AAAI Press, 1998.

[Smith *et al.*, 1998b] Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer Bridge: A Big Win for AI Planning. *AI Magazine*, 19(2):93–105, 1998.

[Son *et al.*, 2001] T.C. Son, C. Baral, and S. McIlraith. Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In Th. Eiter, M. Truszczyński, and W. Faber, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, pages 226–239, Berlin, September 2001. Springer.

[Tate *et al.*, 1994] A. Tate, B. Drabble, and R. Kirby. O-Plan2: An Architecture for Command, Planning and Control. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, volume 1, pages 213–239, San Francisco, CA, 1994. Morgan-Kaufmann.

[Tate, 1977] A. Tate. Generating project networks. In R. Reddy, editor, *Proc. IJCAI-77*, pages 888–893, Boston, MA, USA, 1977. Morgan Kaufmann.

[Turner, 1997] H. Turner. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *The Journal of Logic Programming*, 31(1-3):245–298, 1997.

[Wilkins, 1988] D.E. Wilkins. *Practical Planning - Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, 1988.

[Wilkins, 1990] D. Wilkins. Can AI Planners Solve Practical Problems? *Computational Intelligence*, 6(4):232–246, 1990.

# INDEX