Planning with different forms of domain-dependent control knowledge – an answer set programming approach

Tran Cao Son[†] Chitta Baral[*] Sheila McIlraith [‡]

[†] Department of Computer Science, New Mexico State University
PO Box 30001, MSC CS
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

[*] Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287, USA
chitta@asu.edu

[‡] Knowledge Systems Laboratory,
Computer Science, Stanford University
Stanford, CA 94305, USA
sam@ksl.stanford.edu

Abstract. In this paper we present a declarative approach to adding domain-dependent control knowledge for Answer Set Planning (ASP). Our approach allows different types of domain-dependent control knowledge such as hierarchical, temporal, or procedural knowledge to be represented and exploited in parallel, thus combining the ideas of control knowledge in HTN-planning, GOLOG-programming, and planning with temporal knowledge into ASP. To do so, we view domain-dependent control knowledge as sets of independent constraints. An advantage of this approach is that domain-dependent control knowledge can be modularly formalized and added to the planning problem as desired. We define a set of constructs for constraint representation and provide a set of domain-independent logic programming rules for checking constraint satisfaction.

1 Introduction

Planning is hard. The complexity of classical planning is known to be PSPACE-complete for finite domains and undecidable in the general case [8, 12]. By fixing the length of plans, the planning problem reduces to NP-complete or worse. Planning systems such as FF [16], HSP [6], Graphplan [5], and Blackbox [18] have greatly improved the performance of their systems on benchmark planning problems by exploiting domain-independent search heuristics, clever encodings of knowledge, and efficient data structures [30]. Nevertheless, despite impressive improvements in performance, there is a growing belief that planners that exploit *domain-dependent* control knowledge may provide the key to future performance gains [30]. This conjecture is supported by the impressive performance of planners such as TLPlan [1], TALplan [11] and SHOP [26], all of which exploit domain-dependent control knowledge.

A central issue in incorporating domain-dependent control knowledge into a planner is to identify the classes of knowledge to incorporate and to devise a means of representing and reasoning with this knowledge. In the past, planners such as TLPlan and TALplan have exploited domain-dependent *temporal knowledge*; SHOP and various hierarchical task network (HTN) planners have exploited domain-dependent *hierarchical and partial-order knowledge*; and satisfiability-based planners such as Blackbox have experimented with a variety of domain-dependent control knowledge encoded as propositional formulae. In this paper, we propose to exploit temporal knowledge and hierarchical knowledge as well as, what we refer to as, *procedural knowledge* within the paradigm of answer set planning. We show how these classes of domain-dependent control knowledge can be represented using a normal logic program and how they can be exploited by a basic answer set planner. We demonstrate the improvement in the efficiency of our answer set planner.

The set of programming language constructs provided by the logic programming language GOLOG (e.g., sequence (;), if-then-else, while, etc.) [20] provides an example of the class of procedural knowledge we incorporate into our planner. For example, a procedural constraint written as $a_1; a_2; (a_3|a_4|a_5); f?$ tells the planner that it should make a plan where a_1 is the first action, a_2 is the second action and then it should choose one of a_3, a_4 or a_5 such that after their execution f will be true. This type of domain-dependent control knowledge is different from temporal knowledge where plans are restricted to action sequences that agree with a given set of temporal formulas. Procedural knowledge is also different from hierarchical and partial-order constraints where tasks are divided into smaller tasks, with some partial ordering and other constraints between them. These three classes of domain-dependent control knowledge differ in their structure and while there may be transformations available between one form and another, it is often natural for a user to express knowledge in a particular form.

To exploit the above classes of domain-dependent planning constraints we use the declarative problem-solving paradigm exemplified by satisfiability-based planners. We refer to such an approach to planning as *model-based planning*, to indicate that plans are *models* of the logical theory describing the planning problem. One advantage of this approach is that planner development is divided into two parts: development of model generators for logical languages, and planner encoding as a logical theory. This enables those developing logical encodings of model-based planning problems to exploit the diversity of domain-independent model generators being developed for different tasks.

In this paper, we use an answer set programming appraoch to model-based planning. We use logic programming as the logical language to encode our model-based planning problem. From a knowledge representation perspective, there are many advantages to a logic programming encoding, as compared to a simple propositional logic encoding. These include: parsimonious encoding of solutions to the frame problem in the presence of qualification and ramification constraints; the presence of the non-classical '$\leftarrow$' operator that not only helps in encoding causality but also can be exploited when searching for models; and many fundamental theoretical results [3] that help construct proofs of the correctness of encodings. In contrast, few of the encodings of satisfiability-based planners have proofs of correctness, while most logic programming encodings are accompanied by a proof of correctness. From the perspective of computation, planners based on propositional encodings still fare better. There are currently more implementations of propositional solvers than of logic programming answer set generators, and the best propositional solvers tend to be faster than the best answer set generators.

The rest of this paper is organized as follows: we will review the basics of action language and answer set planning in the next section. We then introduce different constructs for domain-dependent control knowledge representation. For each construct, we provide a set of logic programming rules as its implementation (Subsections 3.1-3.3). We use Smodels, an implemented system for computing stable models of logic programs [27], in our experiments. As such, the rules developed in this paper are written in Smodels syntax and can be used as input to Smodels program[1]. In Subsection 3.4, we describe some experimental results and conclude in Section 4.

2 Preliminaries

2.1 Action Theories

We use the high-level action description language $\mathcal{B}$ of [15] to represent action theories. In such a language, an action theory consists of two finite, disjoint sets of names called *actions* and *fluents*. Actions transition the system from one state to another. Fluents are propositions whose truth value can change as the result of actions. Unless otherwise stated, a is used to denote an action. f and p are used to denote fluents. The action theory also comprises a set of propositions of the following form:

$$\textbf{caused}(\{p_1, \ldots, p_n\}, f) \tag{1}$$

$$\textbf{causes}(a, f, \{p_1, \ldots, p_n\}) \tag{2}$$

$$\textbf{executable}(a, \{p_1, \ldots, p_n\}) \tag{3}$$

$$\textbf{initially}(f) \tag{4}$$

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$, written as $neg(g)$) and a is an action. (1) represents a static causal law, i.e., a ramification constraint. It conveys that whenever the fluent literals $p_1, \ldots, p_n$ hold, so does f. (2), referred to as a *dynamic causal law*, represents the (conditional) effect of a. Intuitively, a proposition of the form (2) states that f is guaranteed to be true after the execution of a in any state of the world where $p_1, \ldots, p_n$ are true. (3) captures an *executability condition* of a. It says that a is executable in a state in which $p_1, \ldots, p_n$ hold. Finally, propositions of the form (4) are used to describe the initial state. (4) states that f holds in the initial state.

An *action theory* is a pair (D, Γ) where D consists of propositions of the form (1)-(3) and Γ consists of propositions of the form (4). For the purpose of this paper, it suffices to note that the semantics of such an action theory is given by a transition graph, represented by a relation t, whose nodes are the alternative (complete) states of the action theory and whose links (labeled with actions) represent the transition between

[1] Although we use Smodels, we believe that the code presented here could easily be used with DLV [9], following simple modifications to reflect differences in syntax.

its states (see details in [15]). That is, if $\langle s, a, s'\rangle \in t$, then there exists a link with label a from state s to state s'.

A *trajectory* of the system is denoted by a sequence $s_0 a_1 s_1 \ldots a_n s_n$ where s_i's are states and a_i's are actions and $\langle s_i, a_{i+1}, s_{i+1}\rangle \in t$ for $i \in \{0, \ldots, n-1\}$. $s_0 a_1 s_1 \ldots a_n s_n$ is a trajectory of a fluent formula Δ if Δ holds in s_n.

In this paper, we will assume that Γ is *complete*, i.e., for every fluent f, either **initially**(f) or **initially**$(neg(f))$ belongs to Γ. We will also assume that (D, Γ) is *consistent* in the sense that there exists a non-empty relation t representing the transition graph of (D, Γ).

2.2 Answer Set Planning

A *planning problem* is specified by a triple $\langle D, \Gamma, \Delta\rangle$ where (D, Γ) is an action theory and Δ is a fluent formula (or *goal*), representing the goal state. A sequence of actions $a_1, \ldots, a_m$ is a *possible plan for* Δ if there exists a trajectory $s_0 a_1 s_1 \ldots a_m s_m$ such that s_0 and s_m satisfy Γ and Δ, respectively[2].

Given a planning problem $\langle D, \Gamma, \Delta\rangle$, answer set planning solves it by translating it into a logic program $\Pi(D, \Gamma, \Delta)$ (or Π, for short) consisting of *domain-dependent* rules that describe D, Γ, and Δ respectively, and *domain-independent* rules that generate action occurrences and represent the transitions between states.

• **Goal representation.** To encode Δ, we define formulas and provide a set of rules for formula evaluation. We consider formulas that are bounded classical formulas with each bound variable associated with a sort. They are formally defined as follows.

– A literal is a formula.
– The negation of a formula is a formula.
– A finite conjunction of formulas is a formula.
– A finite disjunction of formulas is a formula.
– If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f_1(X_1, \ldots, X_n)$ is a formula then $\forall X_1, \ldots, X_n.f_1(X_1, \ldots, X_n)$ is a formula.
– If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f_1(X_1, \ldots, X_n)$ is a formula then $\exists X_1, \ldots, X_n.f_1(X_1, \ldots, X_n)$ is a formula.

A sort called *formula* is introduced and each non-atomic formula is associated with a unique name and defined by (possibly) a set of rules. For example, the conjunction $f \wedge g \wedge h$ is represented by the set of atoms $\{conj(f'), in(f, f'), in(g, f'), in(h, f')\}$ where f' is the name assigned to $f \wedge g \wedge h$; $\forall X_1, \ldots, X_n.f_1(X_1, \ldots, X_n)$ can be represented by the rule

$$formula(forall(f, f_1(X_1, \ldots, X_n))) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n)$$

[2] Note that the notion of plan employed here is weaker than the conventional one where the goal must be achieved on every possible trajectory. This is because an action theory with causal laws can be non-deterministic. Note however, that if D is deterministic, i.e., for every pair (s,a) there exists at most one state s' such that $\langle s, a, s'\rangle \in t$, then every possible plan for Δ is also a plan for Δ.

where f is the name assigned to the formula. In keeping with previous notation, negation is denoted by the function symbol neg. For example, if f is the name of a formula then $neg(f)$ is a formula denoting its negation. Rules to check when a formula holds or does not hold can be written in a straightforward manner and are omitted here to save space. (Details can be downloaded from the Web[3].)

• **Action theory representation.** Since each set of literals $\{p_1, \ldots, p_n\}$ in (1)-(3) can be represented by a conjunction of literals, D can be encoded as a set of facts of Π as follows. First, we assign to each set of fluent literals that occurs in a proposition of D a distinguished name. The constant nil denotes the set $\{\}$. A set of literals $\{p_1, \ldots, p_n\}$ will be replaced by the set of atoms $Y = \{conj(s), in(p_1, s), \ldots, in(p_n, s)\}$ where s is the name assigned to $\{p_1, \ldots, p_n\}$. With this representation, propositions in D can be easily translated into a set of facts of Π. For example, a proposition $causes(a, f, \{p_1, \ldots, p_n\})$ with $n > 0$ is encoded as a set of atoms consisting of $causes(a, f, s)$ and the set Y (s is the name assigned to $\{p_1, \ldots, p_n\}$).

• **Domain independent rules.** The domain independent rules of Π are adapted mainly from [14, 10, 21, 22]. The main predicates in these rules are:

– $holds(L, T)$: L holds at time T,
– $possible(A, T)$: action A is executable at time T,
– $occ(A, T)$: action A occurs at time T, and
– $hf(\varphi, T)$: formula φ holds at time T.

The main rules are given next. In these rules, T is a variable of the sort $time$, L, G are variables denoting *fluent literals* (written as F or $neg(F)$ for some fluent F), S is a variable set of the sort $conj$ (conjunction), and A, B are variables of the sort $action$.

$$holds(L, T{+}1) \leftarrow occ(A, T), causes(A, L, S), hf(S, T). \quad (5)$$
$$holds(L, T) \leftarrow caused(S, L), hf(S, T). \quad (6)$$
$$holds(L, T{+}1) \leftarrow contrary(L, G), holds(L, T), not\ holds(G, T{+}1). \quad (7)$$
$$possible(A, T) \leftarrow executable(A, S), hf(S, T). \quad (8)$$
$$holds(L, 0) \leftarrow literal(L), initially(L). \quad (9)$$
$$nocc(A, T) \leftarrow A \neq B, occ(B, T), T{<}length. \quad (10)$$
$$occ(A, T) \leftarrow T < length, possible(A, T), not\ nocc(A, T). \quad (11)$$

Here, (5) encodes the effects of actions, (6) encodes the effects of static causal laws, and (7) is the inertial rule. (8) defines a predicate that determines when an action can occur and (9) encodes the initial situation. (10)-(11) generate action occurrences, one at a time. We omit most of the auxiliary rules such as rules for defining contradictory literals etc. The source code and examples can be retrieved from our Web site.

Let $\Pi_n(D, \Gamma, \Delta)$ (or Π_n when it is clear from the context what D, Γ, and Δ are) be the logic program consisting of

– the set of domain-independent rules in which the domain of T is $\{0, \ldots, n\}$,
– the set of atoms encoding D and Γ, and

[3] http://www.cs.nmsu.edu/~tson/asp_planner

- the rule $\leftarrow not\ hf(\Delta, n)$ that encodes the requirement that Δ holds at n.

The following result (adapted from [22]) shows the equivalence between trajectories of Δ and stable models of Π_n. Let S be a stable model of Π_n, define $s(i) = \{f \mid holds(f, i) \in S\}$ and $A[i, j] = a_i, \ldots, a_j$ where i or j are integers, f is a fluent, a_t's are actions, and for every t, $i \leq t \leq j$, $occ(a_t, t) \in S$.

Theorem 1. *For a planning problem $\langle D, \Gamma, \Delta \rangle$,*

- *if $s_0 a_0 \ldots a_{n-1} s_n$ is a trajectory of Δ, then there exists a stable model S of Π_n such that $A[0, n-1] = [a_0, \ldots, a_{n-1}]$ and $s_i = s(i)$ for $i \in \{0, \ldots, n\}$, and*
- *if S is a stable model of Π_n with $A[0, n-1] = [a_0, \ldots, a_{n-1}]$ then $s(0) a_0 \ldots a_{n-1} s(n)$ is a trajectory of Δ.*

3 Control Knowledge as Constraints

In this section, we add domain-dependent control knowledge to ASP by viewing it as constraints on the stable models of the program Π. For each type of control knowledge[4], we introduce new constructs for its encoding and present a set of rules that check when a constraint is satisfied.

3.1 Temporal Knowledge

In [1], temporal knowledge is used to prune the search space. Temporal constraints are specified using a linear temporal logic with a precisely defined semantics. It is easy to add them to (or remove them from) a planning problem since their representation is separate from the action and goal representation. Planners exploiting temporal knowledge to control search have proven to be highly efficient and to scale up well [2]. In this paper, we represent temporal knowledge using temporal formulas. In our notation, a temporal formula is either

- a formula (as defined in previous section), or
- a formula of the form $until(\phi, \psi)$, $always(\phi)$, $eventually(\phi)$, or $next(\phi)$ where ϕ and ψ are temporal formulas.

For example, in a logistics domain, let P and L denote a package and its location, respectively. The following formula:

$$always((goal(P, L) \land at(P, L)) \Rightarrow next(\neg holding(P))) \qquad (12)$$

can be used to express that if the goal is to have a package at a particular location and if the package is indeed at that location then it's always the case that the agent will not be holding the package in the next state. This has the effect of preventing the agent from picking up the package once it's at its goal location.

Like non-atomic formulas, temporal formulas can be encoded in ASP using constants, atoms, and rules. For example, the formula $until(f, next(g))$ is represented by

[4] We henceforth abbreviate domain-dependent control knowledge as *control knowledge*.

the set of atoms $\{tf(n_1, next(g)), tf(n_2, until(f, n_1))\}$ where *tf* stands for "temporal formula" and n_1 and n_2 are the new constants assigned to $next(g)$ and $until(f, neg(g))$, respectively. The semantics of these temporal operators is the standard one.

To complete the encoding of temporal constraints, we provide the rules for temporal formula evaluation. The key rules, which define the satisfiability of a temporal formula N at time T ($htf(N,T)$) and between T and T' ($hd(N,T,T')$), are given below.

$$htf(N,T) \leftarrow formula(N), hf(N,T) \tag{13}$$

$$hf(N,T) \leftarrow tf(N,N_1), htf(N_1,T) \tag{14}$$

$$htf(N,T) \leftarrow tf(N, until(N_1,N_2)), hd(N_1,T,T'), htf(N_2,T'). \tag{15}$$

$$htf(N,T) \leftarrow tf(N, always(N_1)), hd(N_1,T,length{+}1). \tag{16}$$

$$htf(N,T) \leftarrow tf(N, eventually(N_1)), htf(N_1,T'), T \leq T'. \tag{17}$$

$$htf(N,T) \leftarrow tf(N, next(N_1)), htf(N_1, T+1). \tag{18}$$

$$not\_hd(N,T,T') \leftarrow not\ htf(N,T''), T{\leq}T''{<}T'. \tag{19}$$

$$hd(N,T,T') \leftarrow htf(N,T), not\ not\_hd(N,T,T') \tag{20}$$

Having defined temporal constraints and specified when they are satisfied, adding temporal knowledge to a planning problem in ASP is easy. We must: (i) encode the knowledge as a temporal formula, say ϕ; (ii) add the rules (13)-(20) to Π; and (iii) add the constraint $\leftarrow\ not\ htf(\phi, 0)$ to Π. Step (iii) eliminates models of Π in which ϕ does not hold. For example, if Π is the program for planning in the logistics domain, adding the constraint (12) to Π will eliminate all models whose corresponding trajectory admits an action occurrence that causes the $holding(P)$ to be true after P is delivered at its destination. As a concrete example, given the goal formula $at(p, l_2)$, there exists no model of Π that corresponds to the sequence of actions $pick\_up(p, l_1), move(l_1, l_2), drop(p, l_2), pick\_up(p, l_2)$. (We appeal to the users for the intuitive meaning of the effects of actions, the initial setting, and the goal of the problem.)

3.2 Procedural Knowledge

Procedural knowledge can be thought of as an (under-specified) sketch of the plans to be generated. This type of control knowledge has been used in GOLOG, an Algol-like logic programming language for agent programming, control and execution, based on a situation calculus theory of actions [20]. GOLOG has been primarily used as a programming language for high-level agent control in dynamical environments (see e.g. [7]). More recently, Golog has been used for general planning [13]. In the planning context, a GOLOG program specifies an arbitrarily incomplete plan that includes non-deterministic choice points that are filled in by the planner (the deductive machinery of a GOLOG-interpreter). For example, a simple GOLOG program $\mathbf{a_1}; \mathbf{a_2}; (\mathbf{a_3}|\mathbf{a_4}|\mathbf{a_5}); \mathbf{f}?$ represents plans which have a_1 followed by a_2, followed by one of a_3, a_4, or a_5 such that f is true upon termination of the plan. The interpreter, when asked for a solution to this program, needs only to decide which one of a_3, a_4, or a_5 it should choose. To encode procedural knowledge, we introduce a set of Algol-like constructs such as sequence, loop, conditional, and nondeterministic choice of arguments/actions. These constructs are used to encode partial procedural control knowledge in the form of programs which are defined inductively as follows. For an action theory (D, Γ) we define a program syntactically as follows.

- an action a is a program,
- a formula ϕ is a program[5],
- if p_i's are programs then $p_1; \ldots ; p_n$ is a program,
- if p_i's are programs then $p_1 | \ldots | p_n$ is a program,
- if p_1 and p_2 are programs and ϕ is a formula then "**if** ϕ **then** p_1 **else** p_2" is a program,
- if p is a program and ϕ is a formula then "**while** ϕ **do** p" is a program, and
- if X is a variable of sort s, $p(X)$ is a program, and $f(X)$ is a formula, then **pick**$(X, f(X), p(X))$ is a program.

As is common practice with Smodels, we will assign to each program a name (with the exception of actions and formulas), provide rules for the construction of programs, and use prefix notation. A sequence $\alpha = p_1; \ldots ; p_n$ will be represented by the atoms $proc(p)$, $head(p, n_1)$, $tail(p, n_2)$ and the set of atoms representing $p_2; \ldots ; p_n$, where p, n_1, and n_2 are the names assigned to α, p_1 (if it is not a primitive action or a formula), and $p_2; \ldots ; p_n$, respectively.

The operational semantics of programs specifies when a trajectory $s_0 a_0 s_1 \ldots a_{n-1} s_n$, denoted by α, is *a trace of a program p* and is defined as follows.

- for $p = a$ and a is an action, $n = 1$ and $a_0 = a$,
- for $p = \phi$, $n = 0$ and ϕ holds in s_0,
- for $p = p_1; p_2$, there exists an i such that $s_0 a_0 \ldots s_i$ is a trace of p_1 and $s_i a_i \ldots s_n$ is a trace of p_2,
- for $p = p_1 | \ldots | p_n$, α is a trace of p_i for some $i \in \{1, \ldots, n\}$,
- for $p =$ **if** ϕ **then** p_1 **else** p_2, α is a trace of p_1 if ϕ holds in s_0 or α is a trace of p_2 if $neg(\phi)$ holds in s_0,
- for $p =$ **while** ϕ **do** p_1, $n = 0$ and $neg(\phi)$ holds in s_0 or ϕ holds in s_0 and there exists some i such that $s_0 a_0 \ldots s_i$ is a trace of p_1 and $s_i a_i \ldots s_n$ is a trace of p, and
- for $p =$ **pick**$(X, f(X), q(X))$, then there exists a constant x of the sort of X such that $f(x)$ holds in s_0 and α is a trace of $q(x)$.

The logic programming rules that realize this semantics follow. We define a predicate $trans(p, t_1, t_2)$ which holds in a stable model S iff $s(t_1) a_{t_1} \ldots s(t_2)$ is a trace of p[6].

$$trans(P, T_1, T_2) \leftarrow proc(P), head(P, P_1), tail(P, P_2), trans(P_1, T_1, T_3), trans(P_2, T_3, T_2). \quad (21)$$

$$trans(A, T, T+1) \leftarrow action(A), A \neq null, occ(A, T). \quad (22)$$

$$trans(null, T, T) \leftarrow \quad (23)$$

$$trans(N, T_1, T_2) \leftarrow choiceAction(N), in(P_1, N), trans(P_1, T_1, T_2). \quad (24)$$

[5] This is analogous to the GOLOG test action $f?$ which tests the truth value of a fluent.

[6] Recall that we define $s(i) = \{holds(f, i) \in S \mid f \text{ is a fluent}\}$ and assume $occ(a_i, i) \in S$.

$$
\begin{aligned}
trans(F, T_1, T_1) &\leftarrow formula(F), hf(F, T_1). && (25)\\
trans(I, T_1, T_2) &\leftarrow if(I, F, P_1, P_2), && (26)\\
&\quad\; hf(F, T_1), trans(P_1, T_1, T_2).\\
trans(I, T_1, T_2) &\leftarrow if(I, F, P_1, P_2), && (27)\\
&\quad\; not\ hf(F, T_1), trans(P_2, T_1, T_2).\\
trans(W, T_1, T_2) &\leftarrow while(W, F, P), hf(F, T_1), T_1 \le T_3 \le T_2, && (28)\\
&\quad\; trans(P, T_1, T_3), trans(W, T_3, T_2).\\
trans(W, T, T) &\leftarrow while(W, F, P), not\ hf(F, T). && (29)\\
trans(S, T_1, T_2) &\leftarrow choiceArgs(S, F, P), && (30)\\
&\quad\; hf(F, T_1), trans(P, T_1, T_2).
\end{aligned}
$$

Finding a valid instantiation of a program P can be viewed as a planning problem $\langle D, \Gamma, \Delta\rangle$ where Δ is the constraint $\leftarrow not\ trans(P, 0, n)$. Let Π_n^T be the program obtained from Π_n by (i) adding the rules (21)-(30), and (ii) replacing the goal constraint with $\leftarrow not\ trans(P, 0, n)$. The following theorem is similar to Theorem 1.

Theorem 2. *Let (D, Γ) be an action theory and P be a program. Then, (i) for every stable model S of Π_n^T, $s(0)a_0 \dots a_{n-1}s(n)$ is a trace of P; and (ii) if $s_0a_0 \dots a_{n-1}s_n$ is a trace of P then there exists a stable model S of Π_n^T such that $s_j = s(j)$ and $occ(a_i, i) \in S$ for $j \in \{0, \dots, n\}$ and $i \in \{0, \dots, n-1\}$.*

3.3 HTN Knowledge

GOLOG programs are good for representing procedural knowledge but prove cumbersome for encoding partial orderings between programs and do not allow temporal constraints. For example, to represent that any sequence containing the n programs $p_1, \dots, p_n$, in which p_1 occurs before p_2, is a valid plan for a goal Δ, one would need to list all the possible sequences and then use the non-deterministic construct[7]. This can be easily represented by an HTN consisting of the set $\{p_1, \dots, p_n\}$ and a constraint expressing that p_1 must occur before p_2. HTNs also allows maintenance constraints of the form $always(\phi)$ to be represented. However, HTNs do not have complex constructs such as procedures, conditionals, or loops. Attempts to combine hierarchical constraints and GOLOG-like programs (e.g., [4]) have fallen short since they do not allow complex programs to occur within these HTN programs. We will show next that, under the ASP framework, this restriction can be eliminated by adding the following item to the definition of programs in the previous section.

– If $p_1, \dots, p_n$ are programs then a pair (S, C) is a program where $S = \{p_1, \dots, p_n\}$ and C is a set of ordering or truth constraints (defined below).

Let $S = \{p_1, \dots, p_k\}$ be a set of programs. Assume that n_i, $1 \le i \le k$, is the name assigned to the program p_i. An ordering constraint over S has the form $n_i \prec n_j$

[7] For $n = 3$, the three possibilities are $p_1; p_2; p_3$, $p_1; p_3; p_2$, and $p_3; p_1; p_2$. Using a *concurrent construct* $||$, these three programs can be packed into two programs $p_1; p_2 || p_3$ and $p_1; p_3; p_2$.

where $n_i \neq n_j$ and a truth constraint is of the form (n_i, ϕ), (ϕ, n_i), or (n_i, ϕ, n_t) where ϕ is a formula. In our encoding, we will represent a program (S, C) by an atom $htn(p, Sn, Cn)$ where p, Sn, and Cn are the names assigned to (S, C), S, and C respectively. To complete our extension, we need to define when a trajectory is a trace of a program with the new construct and provide logic program rules for checking its satisfaction. A trajectory $s_0a_0 \ldots a_{n-1}s_n$ is a trace of a program (S, C) if there exists a sequence $j_0{=}0 \leq j_1 \leq \ldots \leq j_k{=}n$ and a permutation $(i_1, \ldots, i_k)$ of $(1, \ldots, k)$ such that the sequence of trajectories $\alpha_1 = s_0a_0 \ldots s_{j_1}$, $\alpha_2 = s_{j_1}a_{j_1} \ldots s_{j_2}$, …, $\alpha_k = s_{j_{k-1}}a_{j_{k-1}} \ldots s_n$ satisfies the following conditions:

- for each l, $1 \leq l \leq k$, α_l is a trace of p_{i_l},
- if $n_t < n_l \in C$ then $i_t < i_l$,
- if $(\phi, n_l) \in C$ (or $(n_l, \phi) \in C$) then ϕ holds in the state $s_{j_{l-1}}$ (or s_{j_l}), and
- if $(n_t, \phi, n_l) \in C$ then ϕ holds in $s_{j_t}, \ldots, s_{j_{l-1}}$.

We will extend the predicate $trans$ to allow the new type of programs to be considered. Rules for checking the satisfaction of a program $htn(N, S, C)$ are given next.

$$trans(N, T_1, T_2) \leftarrow htn(N, S, C), not\ nok(N, T_1, T_2). \tag{31}$$

$$1\{begin(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \tag{32}$$

$$1\{end(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \tag{33}$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), in(I, S), T_3 > T_4, begin(N, I, T_3, T_1, T_2), end(N, I, T_4, T_1, T_2). \tag{34}$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), in(I, S), T_3 \leq T_4, begin(N, I, T_3, T_1, T_2), end(N, I, T_4, T_1, T_2), not\ trans(I, T_3, T_4). \tag{35}$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), not\ trans(N, T_1, T_2). \tag{36}$$

In the above rules, the predicates $begin(N, I, T_3, T_1, T_2)$ and $end(N, I, T_4, T_1, T_2)$ are used to record the beginning and the end of the program I, a member of N. Rules (32)-(33) make sure that each program will have start and times. These two rules are not logic programming rules but are unique to Smodels encodings. They were introduced to simplify the encoding of choice rules [28], and can be translated into a set of normal logic program rules. The predicate $nok(N, T_1, T_2)$ states that the assignments for programs are not acceptable. (We omit the rules that check for the satisfiability of constraints in C of a program $htn(N, S, C)$. They can be downloaded from our Web site.) Theorem 2 will still hold.

3.4 Demonstration Experiments

We tested our implementation with some domains from the general planning literature and from the AIPS planning competition [2]. We chose problems for which procedural control knowledge appeared to be easier to exploit than other types of control knowledge. Our motivation was: (i) it has already been established that well-chosen temporal and hierarchical constraints will improve a planner's efficiency; (ii) we have previously experimented with the use of temporal knowledge in the ASP framework [29]; and (iii) we are not aware of any empirical results indicating the utility of procedural knowledge in planning, especially in ASP. ([13] concentrates on using GOLOG to do planning in domains with incomplete information, not on exploiting procedural knowledge in planning.)

We selected the elevator example from [20] (elp1-elp3) and the Miconic-10 elevator domain (s1-0,…,s5-0s2), proposed by Schindler Lifts Ltd. for the AIPS 2000 competition [2]. Note that some of the planners, that competed in AIPS 2000, were unable to solve this problem. Due to the space limitation we cannot present the action theories and the Smodels encoding of the programs here. They can be found at the URL mentioned previously. The time taken to compute one model with and without control knowledge are given in column 5 and 6 of the table below, respectively.

Problem	Plan Length	# Person	# Floors	With Control Knowledge	Without Control Knowledge
elp1	10	2	6	0.600	**0.560**
elp2	14	3	6	1.411	6.729
elp3	18	4	6	3.224	120.693
s1-0	4	1	2	0.100	**0.020**
s2-0	8	2	4	1.802	**0.921**
s3-0	12	3	6	22.682	34.519
s4-0	15	4	8	164.055	314.101
s5-0s1	19	5	4	57.952	$>$ 2 hours
s5-0s2	19	5	5	105.040	$>$ 2 hours

As can be seen, the encoding with control knowledge yields substantially better performance in situations where the minimal plan length is great. For large instances (the last two rows), Smodels can find a plan using control knowledge in a short time and cannot find a plan in 2 hours without control knowledge. In some small instances (the time in column 6 is in boldface), the speed up cannot make up for the overhead needed in grounding the control knowledge. The output of Smodels for each run is given in the file *result* at the above URL. For larger instances of the elevator domain [2] (5 persons or more and 10 floors or more), our implementation terminated prematurely with either a stack overflow error or a segmentation fault error[8].

[8] Experiments were run on a an HP OmniBook 6000 laptop with 130,544 Kb Ram and an Intel Pentium III 600 MHz processor).

4 Discussions and Future Work

In this paper we presented a declarative approach to adding domain-dependent control knowledge to ASP. Our approach enables different types of control knowledge such as hierarchical, temporal, or procedural knowledge to be represented and exploited in parallel; thus combining the ideas of HTN-planning, GOLOG-programming, and planning with temporal knowledge into ASP. For example, one can find a valid instantiation of a GOLOG program that satisfies some temporal constraints. This distinguishes our work from other related work [17, 19, 4, 25] where only one or two types of constraints were considered or combined. Moreover, in a propositional environment, ASP with procedural knowledge can be viewed as an off-line interpreter for a GOLOG program. Because of the declarative nature of logic programming the correctness of this interpreter is easier to prove than an interpreter written in Prolog. We view domain-dependent control knowledge as independent sets of constraints. An advantage of this approach is that domain-dependent control knowledge can be modularly formalized and added to planning problems as desired.

Our experimental result demonstrates that ASP can scale up better with domain-dependent control knowledge. In keeping with the experience of researchers who have incorporated control knowledge into SATplan (e.g., [19]), we do not expect ASP with only one type of domain-dependent knowledge to do better than TLPLAN [1], as Smodels is a general purpose system. But in the presence of near deterministic procedural constraints, our approach may do better. More rigorous experimentation with a variety of domains including those used in the AIPS planning competition will be a significant focus of our future work.

Acknowledgements The first two authors would like to acknowledge the support of the NASA grant NCC2-1232. The third author would like to acknowledge the support of NASA grant NAG2-1337. The work of Chitta Baral was also supported in part by the NSF grants IRI-9501577 and NSF 0070463. The work of Tran Cao Son was also supported in part by NSF grant EIA-981072.

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1,2):123–191, 2000.
2. F. Bacchus, H. Kautz, D.E. Smith, D. Long, H. Geffner, and J. Koehler. AIPS-00 Planning Competition, http://www.cs.toronto.edu/aips2000/.
3. C. Baral. *Knowledge Representation, reasoning, and declarative problem solving with Answer sets (Book draft)*. 2001.
4. C. Baral and T. C. Son. Extending ConGolog to allow partial ordering. In *ATAL, LNCS, Vol. 1757*, pages 188–204, 1999.
5. A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.
6. B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence - Special issue on Heuristic Search*, 129(1-2):5-33, 2001.
7. W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, Schulz D., W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *AAAI-98*, pages 11–18, 1998.

8. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:161–204, 1994.
9. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and application frontends. In *Proceedings of the 12th Workshop on Logic Programming*, pages 128–137, 1997.
10. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of European Conference on Planning*, pages 169–181, 1997.
11. P. Doherty and J. Kvarnstom. TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner, TIME'99, 1999.
12. K. Erol, D. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.
13. A. Finzi, F. Pirri, and R. Reiter. Open world planning in the situation calculus. In *AAAI-00*, page 754–760, 2000.
14. M. Gelfond. Posting on TAG-mailing list, 1999.
15. M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 3(6), 1998.
16. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253-302, 2001.
17. Y.C. Huang, B. Selman, and H. Kautz. Control knowledge in planning: Benefits and trade-offs. In *AAAI-99*, pages 511–517, 1999.
18. H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop Planning as Combinatorial Search*, AIPS-98.
19. H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of AIPS*, 1998.
20. H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, 1997.
21. V. Lifschitz. Answer Set Planning. In *ICLP'99*, pages 23–37, 1999.
22. V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *LP-NMR'99*, pages 92–106, 1999.
23. A. Lotem and S. Dana Nau. New advances in GraphHTN: Identifying independent subproblems in large HTN domains. In *AIPS*, pages 206–215, 2000.
24. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.
25. S. McIlraith. Modeling and programming devices and web agents. In *Proceedings of the NASA Goddard Workshop on Formal Approaches to Agent-Based Systems, LNCS*, 2000.
26. D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *AAAI-99*, pages 968–973, 1999.
27. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *ICLP & LPNMR*, pages 420–429, 1997.
28. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics for weight constraint rules. In *LPNMR'99*, pages 315–332, 1999.
29. L. Tuan and C. Baral. Effect of knowledge representation on model based planning: experiments using logic programming encodings. In *AAAI Spring Symposium on "Answer Set Programming"*, pages 110–115, 2001.
30. D. Wilkins and M. desJardines. A call for knowledge-based planning. *AI Magazine*, 22(1):99–115, Spring 2001.