Domain-Dependent Knowledge in Answer Set Planning

Tran Cao Son* Chitta Baral† and Tran Hoai Nam† Sheila McIlraith‡

*Computer Science
New Mexico State University
PO Box 30001, MSC CS
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

†Computer Science and Engineering
Arizona State University
Tempe, AZ 85287, USA
{*chitta,namtran*}*@asu.edu*

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

October 13, 2002

Abstract

In this paper we consider three different kinds of domain dependent control knowledge (temporal, procedural and HTN-based) that are useful in planning. Our approach is declarative and relies on the language of logic programming with answer set semantics (LPASS). We show that the addition of these three kinds of control knowledge only involves adding a few more rules to a planner written in LPASS that can plan without any control knowledge. Thus domain dependent control knowledge can be modularly added to (or removed from) a planning problem without the need of modifying the planner. We formally prove the correctness of our planner, both in the absence and presence of the control knowledge. Finally, we do some initial experimentation that shows the reduction in planning time when procedural domain knowledge is used and the plan length is big.

1 Introduction and Motivation

The simplest formulation of planning – referred to as classical planning – entails finding a sequence of actions that takes a world from a completely known initial state to a state that satisfies certain goal conditions. The inputs to a corresponding planner are the descriptions (in a compact description language such as STRIPS [16]) of the effect of actions on the world, the description of the initial state and the description of the goal conditions, and the output is a plan (if it exists) consisting of a sequence of actions. The complexity of classical planning is known to be PSPACE-complete for finite domains and undecidable in the general case [9, 15]. By making certain assumptions such as fixing the length of plans, and requiring actions to be deterministic the complexity reduces to NP-complete.

To be able to plan is widely recognized as an important characteristic of an intelligent entity. Thus when developing many intelligent systems, despite the complexity, we need to be able to build planners. But we can not wish away the complexity. Since the complexity is due to the exponential size of the search space, any approach to overcome the complexity would require searching efficiently and intelligently. This is the crux of the three main successful approaches to planning: (i) using heuristics [7, 21, 6] that are derived from the description, (ii) translating the planning problem to model finding in a logic and using intelligent model finding techniques for that logic [23], and (iii) using domain dependent control knowledge[1] [1, 14, 33]. Among these the third one has led to a set of successful and competitive planners (for example, TLPlan [1], TALplan [14] and SHOP [33]) with respect to several planning benchmarks and is widely predicted [44] to be the most scalable. It should be noted that, strictly speaking, by using the third approach we move away from classical planning as we need an additional input, the domain knowledge.

[1]This is alternatively referred to in the literature as 'domain dependent knowledge', 'control knowledge', 'domain knowledge', and 'domain constraints'. We also sometimes use these shortened terms in this paper.

In this paper our main focus is the third approach of planning using domain knowledge. We discuss various kinds of domain knowledge (that can be exploited by a planner), how to represent such knowledge, and how to use them during planning. In addition, we integrate the second and the third approach as 'logic' being a good tool to express knowledge is a common thread between the two approaches.

By planning using domain knowledge we mean that there is a planner which can, if necessary, plan without any domain knowledge; but can usually plan faster using domain knowledge. Our view is in agreement with the planning systems TLPlan [1], and TALplan [14] but is in disagreement with most HTN planners. The TLPlan and TALplan systems explore the use of temporal domain knowledge in planning. In this paper we identify two other kinds of domain knowledge, procedural and partial ordered, and explore their role in planning.

Our formulation of procedural domain knowledge is inspired by GOLOG, referred to alternatively as a logic programming language, or an action execution language. Although syntactically our procedural domain knowledge is very similar to GOLOG programs, our use of procedural knowledge during planning is very different from how GOLOG programs are used. Similarly, our formulation of partial ordered domain knowledge is inspired by HTN constructs, but our use of this type of knowledge during planning is very different from the workings of HTN planners. The main difference is that typical GOLOG programming or HTN planning do not have an independent planner that can do classical planning without using the knowledge encoded in the GOLOG programs or the HTN descriptions. In our approach, which is similar to the approach in [1], the planner module is separate from the domain knowledge (encoding temporal, procedural, or partial ordered knowledge), and can plan independent of the domain knowledge.

To achieve our goal of planning using domain knowledge, an important first step is to be able to both reason about actions and their effects on the world, and represent and reason about domain knowledge. This leads to the question of choosing an appropriate language for both reasoning and representation tasks. For this we choose the language of logic programming with answer set semantics (LPASS) [19], also referred to as A-Prolog [18] or AnsProlog [4]. The reasons behind our choice of this language over others are many[2] and include:

- LPASS is a non-monotonic language and hence suitable for knowledge representation, especially for reasoning in presence of incomplete knowledge.
- LPASS is more expressive than classical (monotonic) logics such as propositional logic and first-order logic. For example, it can express transitive closure. Also, the non-classical constructs '$\leftarrow$' and ' *not* ' of LPASS allow it to encode a form of directionality that makes it easy to encode causality, which as shown in [29] can not be expressed in classical logic in a straightforward way.
- The non-classical constructs also give a structure to LPASS programs and statements, such as a head and a body, which allows us to define various subclasses each with different complexity and expressibility properties [11]. A particular subclass (when no classical negation is allowed) has the same complexity as propositional logic but more expressibility while the most general case – which allows "**or**" in the head – has the complexity and expressibility of the seemingly more complicated default logic [37]. In general, LPASS is syntactically simpler to other non-monotonic logics and at the same time as expressive as some [37].
- There exists a sizable body of "building block" results about LPASS which makes it more amenable for knowledge representation and for correctness analysis of the representations.

  This includes result about composition of several LPASS programs so that certain original conclusions are preserved (referred to as 'restricted monotonicity'), a transformation of a program so that it can deal with incomplete information, abductive assimilation of new knowledge, language independence and tolerance, splitting an LPASS program to smaller components for computing its

[2]The book [4] contains elaboration of these points.

answer sets, and proving properties about the original program. To the best of our knowledge, for no other knowledge representation language has a comparable body of mathematical results been accumulated.

- There exist several efficient LPASS interpreters [35, 10] and LPASS has been shown to be useful in several application domains other than knowledge representation and planning. This includes policy description, product configuration, cryptography and encryption, wire routing, decision support in a space shuttle and its 'if'–'then' structure has been found to be intuitive for knowledge encoding from a human expert point of view.

- Finally, LPASS has already been used in planning [41, 13, 26], albeit in the absence of domain knowledge. In this regard LPASS is suitable for concisely expressing the effect of actions and static causal relations between fluents. Note that concise expression of effect of actions involves representation of the 'frame problem' which was one of the original motivation behind the development of non-monotonic logics. Together with its ability to enumerate possible action occurrences LPASS is a suitable candidate for model based planning, and falls under the category (ii) of successful approaches to planning.

As evident from our choice of language, our main focus in this paper is the knowledge representation aspects of planning using domain knowledge. In particular, our concern includes:

- the ease of expressing effects of actions on the world, and reasoning about them,
- the ease of expressing and reasoning about various kinds of domain constraints,
- the ease of adding new kind of domain constraints, and
- proving correctness results about the LPASS representation of the planning with domain constraints task.

We do performs some limited efficiency experiments, but leave more detailed experimentation to future work.

With the above focus, the contributions of the paper and the sections they appear in can be summarized as follows:

1. In Section 3 we encode planning (without domain constraints) using LPASS in presence of both dynamic effect of actions and static causal laws, and with goals as restricted first order formulas. We then formally prove the relation between valid trajectories of the action theory, and answer sets of the encoded program. The main difference between our formulation and earlier LPASS encodings [41, 13, 26] is our use of static causal laws, and more general goals, and our consideration of trajectories instead of plans. The reason we relate trajectories instead of plans is because in presence of static causal laws the effect of actions may be non-deterministic.

2. In Section 4.1 we show how to incorporate the use of temporal constraints in planning to the initial planning formulation described in the previous item. The incorporation involves only the addition of a few more rules, thus illustrating the declarativeness and elaboration tolerance of our approach. We then formally prove the relation between valid trajectories of the action theory satisfying the temporal constraints, and answer sets of the updated program. Our approach differs from [1, 14] in that we use LPASS for both the basic encoding of planning and the temporal constraints, while the planners in [1, 14] are written in procedural languages. Preliminary experiments show our approach to be less efficient. But our use of LPASS allows us to have correctness proofs, which is one of our major concerns. Such correctness proofs are not part of [1, 14].

3. In Section 4.2 we consider the use of procedural domain knowledge in planning. An example of a procedural domain knowledge is a program written as $a_1; a_2; (a_3|a_4|a_5); f?$. This program 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 the plan's execution f will be true.

   We define programs representing procedural domain knowledge and specify when a trajectory is a trace of such a program. We then show how to incorporate the use of procedural domain knowledge in planning to the initial planning formulation described in item (1.). As in (2.) the incorporation involves only the addition of a few more rules. We then formally prove the relation between valid trajectories of the action theory satisfying the procedural domain knowledge, and answer sets of the updated program. We also present experimental results (Section 4.4) showing the improvement in planning time due to using such knowledge over planning in the absence of such knowledge.

4. In Section 4.3 we motivate the need of additional constructs from HTN-planning to express domain knowledge and integrate features of HTN with procedural constructs to develop a more general language for domain knowledge. We then define trace of such general programs and show how to incorporate them in planning. We then formally prove the relation between valid trajectories of the action theory satisfying the general programs containing both procedural and HTN constructs, and answer sets of the updated program. To the best of our knowledge this is the first time an integration of HTN and procedural constructs has been proposed for use in planning.

5. As mentioned in the above items, we pay major attention to correctness proofs of our LPASS formulations. All the proofs appear in Appendix A, and for completeness we present a few results about LPASS, that we use in our proofs, in Appendix B.

In regards to closely related work, although planning through model finding of propositional encodings [23] has been studied quite a bit, those papers do not have correctness proofs and do not use the varied domain constraints that we use in this paper.

We now start with some preliminaries and background material about reasoning about actions and LPASS, which will be used in the rest of the paper.

2 Preliminaries and Background

2.1 Reasoning about actions: the action description language $\mathcal{B}$

Recall that planning involves finding a sequence of actions that takes a world from a given initial state to a state that satisfies certain goal conditions. To do planning, we must be first able to reason about the impact of a single action on a world. This is also the first step in 'reasoning about actions'. In general, reasoning about actions involves defining a transition function from states (of the world) and actions to sets of states where the world might be after executing the action. Since explicit representation of this function would require exponential space in the size of the number of fluents (i.e., properties of the world), actions and their effects on the world are described using an action description language, and the above mentioned transition function is implicitly defined in terms of that description.

We now present the action description language $\mathcal{B}$ from [20] which we will use in this paper. This language consists of two finite, disjoint sets of names $\mathbf{A}$ and $\mathbf{F}$, called *actions* and *fluents*, respectively, and a set of propositions of the following form:

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

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

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

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

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$) and a is an action. (1) represents a *static causal law*, i.e., a ramification constraint. It conveys the meaning 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 while (3) states an executability condition 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. An executability condition of a says that a is executable in a state in which $p_1, \ldots, p_n$ hold. Propositions of the form (4) are used to describe the initial state. It states that f holds in the initial state.

An *action theory* is a pair (D, Γ) where Γ, called the *initial state description*, consists of propositions of the form (4) and D, called *the domain description*, consists of propositions of the form (1)-(3). For convenience, we sometimes denote the set of propositions of the form (1), (2), and (3) by D_C, D_D, and D_E, respectively.

Example 1 The well-known blocks world domain can be expressed using the following propositions[3]:

$$D_1 = \left\{ \begin{array}{l}
\mathbf{causes}(move(X,Y), on(X,Y), \{\}), \quad \text{for all } \; X \neq Y \\
\mathbf{causes}(move(X,Y), \neg clear(Y), \{\}), \quad \text{for all } \; X \neq Y \\
\mathbf{causes}(move(X,table), on(X,table), \{\}) \\
\mathbf{caused}(\{on(X,Y)\}, \neg on(Z,Y)), \quad \text{for all } \; Z \neq X \\
\mathbf{caused}(\{on(X,Y)\}, \neg on(X,Z)), \quad \text{for all } \; Z \neq Y \\
\mathbf{caused}(\{on(X,table)\}, \neg on(X,Y)), \quad \text{for all } \; Y \\
\mathbf{caused}(\{\neg on(Y_1,X), \ldots, \neg on(Y_n,X)\}, clear(X)) \quad \text{where } \; Y_i'\text{s are blocks different than } X \\
\mathbf{executable}(move(X,Y), \{clear(X), clear(Y)\}) \\
\mathbf{executable}(move(X,table), \{clear(X)\})
\end{array} \right.$$

where X, Y are variables of type "*block*", "*table*" is a constant. The actions are "$move(X,Y)$" and "$move(X, table)$", which mean moving block X onto block Y and onto the "*table*", respectively. The fluents are "$on(X,Y)$", "$clear(X)$", and "$on(X, table)$". They are used to record the positional information about the blocks.

Let D be the theory for the domain of blocks a, b, c. An example of an initial state could be given by the set of initial propositions:

$$\Gamma = \left\{ \begin{array}{l}
\mathbf{initially}\ on(a,table), \mathbf{initially}\ on(b,a), \mathbf{initially}\ on(c,b), \\
\mathbf{initially}\ \neg on(b,table), \mathbf{initially}\ \neg on(b,c), \\
\mathbf{initially}\ \neg on(a,b), \mathbf{initially}\ \neg on(a,c), \\
\mathbf{initially}\ \neg on(c,table), \mathbf{initially}\ \neg on(c,a), \\
\mathbf{initially}\ \neg clear(a), \mathbf{initially}\ \neg clear(b), \mathbf{initially}\ clear(c)
\end{array} \right\}$$

□

A domain description given in $\mathcal{B}$ defines a transition function from pairs of actions and states to sets of states whose precise definition is given below. Intuitively, given an action a and a state s, the transition function Φ defines the set of states $\Phi(a,s)$ that may be reached after executing the action a in state s. If $\Phi(a,s)$ is an empty set it means that a is not executable in s. We now formally define Φ.

Let D be a domain description in $\mathcal{B}$. An *interpretation* I of the fluents in D is a maximal consistent set of fluent literals drawn from $\mathbf{F}$. A fluent f is said to be true (resp. false) in I iff $f \in I$ (resp. $\neg f \in I$). The truth value of a fluent formula in I is defined recursively over the propositional connectives in the usual way. For example, $f \wedge g$ is true in I iff f is true in I and g is true in I. We say that a formula φ holds in I (or I satisfies φ), denoted by $I \models \varphi$, if φ is true in I.

[3]We follow the convention in logic programming in that terms beginning with a capital and lower-case letters represent variables and constants, respectively. A proposition with variables represents the set of its ground instances.

Let u be a consistent set of fluent literals and K a set of static causal laws. We say that u is closed under K if for every static causal laws "**caused**$(\{p_1, \ldots, p_n\}, f)$" in K, if $\{p_1, \ldots, p_n\} \subseteq u$ then so does f. By $Cl_K(u)$ we denote the least consistent set of literals from D that contains u and is also closed under K.

Formally, a *state* of D is an interpretation of the fluents in $\mathbf{F}$ that is closed under the set of static causal laws D_C of D.

An action a is *executable* in a state s if there exists an executability proposition "**executable**$(a, \{f_1, \ldots, f_n\})$" in D such that $s \models f_1 \wedge \ldots \wedge f_n$. Clearly, if "**executable**$(a, \{\})$" belongs to D, then a is executable in every state of D.

The *direct effect of an action* a in a state s of D is the set $E(a, s) = \{f \mid D$ contains a dynamic law "**causes**$(a, f, \{f_1, \ldots, f_n\})$" and $f_i \in s$ for $i = 1, \ldots, n\}$.

For a domain description D, $\Phi(a, s)$, the set of states that may be reached by executing a in s, is defined as follows.

1. If a is executable in s, then
$$\Phi(a, s) = \{s' \mid s' \text{ is a state and } s' = Cl_{D_C}(E(a, s) \cup (s \cap s'))\};$$
2. If a is not executable in s, then $\Phi(a, s) = \emptyset$.

The intuition behind the above formulation is as follows. The direct effects of an action a in a state s are determined by the dynamic causal laws and are given by $E(a, s)$. All fluent literals in $E(a, s)$ must hold in any resulting state. The set $s \cap s'$ contains the fluent literals of s which continue to hold by inertia, i.e they hold in s' because of not being changed by any action. In addition, the resulting state must be closed under the set of static causal laws D_C. These three aspects are captured by the definition above. Observe that when D_C is empty and a is executable in state s, $\Phi(a, s)$ is equivalent to the set of states that satisfy $E(a, s)$ and are closest to s using symmetric difference[4] as the measure of closeness [31]. Additional explanations and motivations behind the above definition can be found in [3, 31, 43].

Every domain description D in $\mathcal{B}$ has a unique transition function Φ, and we say Φ is the transition function of D. We illustrate the definition of the transition function in the next example.

Example 2 Consider the block word domain from Example 1 with the set of blocks $\{a, b, c, d, e, f\}$. The state depicted in the Fig. 1 is given by the set[5]

$$s_0 = \{on(a, table), on(b, a), on(c, b), clear(c), on(d, table), on(e, d), on(f, e), clear(f)\}.$$

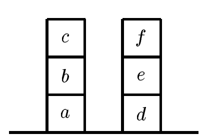

Figure 1: A state of the block world domain with 6 blocks a, b, c, d, e, and f

[4]We say s_1 is strictly closer to s than s_2 if $s_1 \setminus s \cup s \setminus s_1 \subset s_2 \setminus s \cup s \setminus s_2$.

[5]To simplify the notation, we list only positive literals in states (i.e., whatever is not in a state is false).

In state s_0, the actions $move(c, table)$, $move(f, table)$, $move(f, c)$, and $move(c, f)$ are executable. We have the following possible transitions from state s_0:

$$
\begin{array}{rcl}
(s_0 \cup \{\, on(c, table), clear(b) \,\}) \setminus \{\, on(c, b) \,\} & \in & \Phi(move(c, table), s_0). \\
(s_0 \cup \{\, on(f, table), clear(e) \,\}) \setminus \{\, on(f, e) \,\} & \in & \Phi(move(f, table), s_0). \\
(s_0 \cup \{\, on(c, f), clear(b) \,\}) \setminus \{\, on(c, b), clear(f) \,\} & \in & \Phi(move(c, f), s_0). \\
(s_0 \cup \{\, on(f, c), clear(e) \,\}) \setminus \{\, on(f, e), clear(c) \,\} & \in & \Phi(move(f, c), s_0).
\end{array}
$$

□

For a domain description D with transition function Φ, 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 is called a *trajectory* in D if $s_{i+1} \in \Phi(s_i, a_{i+1})$ for $i \in \{0, \ldots, n-1\}$. A trajectory $s_0 a_1 s_1 \ldots a_n s_n$ achieves a fluent formula Δ if $s_n \models \Delta$.

A domain description D is *consistent* iff for every action a and state s, if a is executable in s, then $\Phi(a, s) \neq \emptyset$. An action theory (D, Γ) is consistent if D is consistent and $s_0 = \{f \mid \textbf{initially}(f) \in \Gamma\}$ is a state of D. In what follows, we will consider only consistent action theories.

2.2 Logic Programming with answer set semantics (LPASS) and its application

In this section we review LPASS and and its applicability to problem solving.

2.2.1 Logic Programming with answer set semantics (LPASS)

Although the programming language Prolog and the field of logic programming have been around for several decades, the answer set semantics of logic programs – initially referred to as the stable model semantics, was rather recently proposed by Gelfond and Lifschitz in [19]. Unlike earlier characterizations of logic programs where the goal was to find a unique appropriate 'model' of a logic program, the answer set semantics allows the possibility that a logic program may have multiple appropriate models, or no appropriate models at all. It is this feature of the answer set semantics that plays an important role in using LPASS for problem solving. We now present the syntax and semantics of LPASS.

A logic program Π is a set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{5}$$

or

$$\bot \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{6}$$

where $0 \leq m \leq n$, each a_i is an atom of a first-order language $\mathcal{LP}$, $\bot$ is a special symbol denoting the truth value *false*, and *not* denotes a special unary predicate – the negation-as-failure operator. A negation as failure literal (or naf-literal) is of the form *not* a where a is an atom. For a rule of the form (5)-(6), the left and right hand side of the rule are called the *head* and the *body*, respectively. A rule of the form (6) is also called a constraint.

Given a logic program Π. We will assume that each rule in Π is replaced by the set of its ground instances so that all atoms in Π are ground. Consider a set of ground atoms X. The body of a rule of the form (5) or (6) is satisfied by X if $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$ and $\{a_1, \ldots, a_m\} \subseteq X$. A rule of the form (5) is satisfied by X if either its body is not satisfied by X or $a_0 \in X$. A rule of the form (6) is satisfied by X if its body is not satisfied by X. An atom a is supported by X if a is the head of some rule (5) such that $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$.

For a set of ground atoms X and a program Π, the reduct of Π with respect to X, denoted by Π^X, is the program obtained from the set of all ground instances of Π by deleting

1. each rule that has a naf-literal *not* a in its body with $a \in S$, and
2. all naf-literals in the bodies of the remaining clauses.

S is an *answer set (or a stable model)* of Π if it satisfies the following conditions.

1. If Π does not contain any naf-literal (i.e. $m = n$ in every rule of Π) then S is the smallest set of atoms such that
   (a) for any ground instance $a_0 \leftarrow a_1, \ldots, a_m$ of a rule from Π, if $a_1, \ldots, a_m \in S$, then $a_0 \in S$, and
   (b) for any ground instance $\perp \leftarrow a_1, \ldots, a_m$ of a rule from Π, $\{a_1, \ldots, a_m\} \setminus S \neq \emptyset$.
2. If the program Π does contain some naf-literal ($m < n$ in some rule of Π), then S is an answer set of Π if S is the answer set of Π^S. (Note that Π^S does not contain naf-literals, its answer set is defined in the first item.)

A program Π is said to be *consistent* if it has an answer set. Otherwise, it is inconsistent.

Many robust and efficient systems that can compute answer sets of propositional logic programs have been developed. Two of the frequently used systems are **dlv** [10] and **smodels** [35]. Recently, **XSB** [40], a system developed for computing the well-founded model of logic programs, has been extended to compute stable models of logic programs as well.

2.2.2 Problem solving using LPASS

Prolog and other early logic programming systems were geared towards answering yes/no queries with respect to a program, and if the queries had variable then returning instantiations together with an 'yes' answer. The possibility of multiple answer sets and no answer sets has given rise to an alternative way to solve problems using LPASS. In this approach, referred to by some as answer set programming (also known as stable model programming) [30, 34, 26], possible solutions of a problem are enumerated as answer set candidates and non-solutions are eliminated through rules with $\perp$ in the head, resulting in a program whose answer sets have one-to-one correspondence with the solutions of the problem.

We illustrate the concepts of answer set programming by showing how the 3-coloring problem of a bi-directed graph G can be solved using LPASS. Let the three colors be red (r), blue (b), and green (g) and the vertex of G be $0, 1, \ldots, n$. Let $P(G)$ be the program consisting of

- the set of atoms $edge(u, v)$ for every edge (u, v) of G,
- for each vertex u of G, three rules stating that u must be assigned one of the colors red, blue, or green:

$$color(u, g) \leftarrow not\ color(u, b), not\ color(u, r)$$
$$color(u, r) \leftarrow not\ color(u, b), not\ color(u, g)$$
$$color(u, b) \leftarrow not\ color(u, r), not\ color(u, g)$$

  and

- for each edge (u, v) of G, three rules representing the constraint that u and v must have different color:

$$\perp \leftarrow color(u, r), color(v, r), edge(u, v)$$
$$\perp \leftarrow color(u, b), color(v, b), edge(u, v)$$
$$\perp \leftarrow color(u, g), color(v, g), edge(u, v)$$

It can be shown that for each graph G, (i) $P(G)$ is inconsistent iff the 3-coloring problem of G does not have a solution; and (ii) if $P(G)$ is consistent then each answer set of $P(G)$ corresponds to a solution of the 3-coloring problem of G and vice versa.

To make answer set style programming easier, Niemelä et al. [36] introduce a new type of rules, called *cardinality constraint rule* (a special form of the *weight constraint rule*) of the following form:

$$l\{b_1, \ldots, b_k\}u \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n \tag{7}$$

where a_i and b_j are atoms and l and u are two integers, $l \leq u$. The intuitive meaning of this rule is that whenever its body is satisfied then at least l and at most u atoms of the set $\{b_1, \ldots, b_k\}$ must be true. Using rules of this type, one can greatly reduce the number of rules of programs in answer set programming. For instance, in the above example, the three rules representing the constraint that every node u needs to be assigned one of the three colors can be packed into one cardinality constraint rule:

$$1\{color(u, g), color(u, r), color(u, b)\}1 \leftarrow$$

The semantics of logic programs with such rules is given in [36]. For our purpose in this paper we only need to consider rules with $l = u = 1$, and restrict that if we have rules of the form (7) in our program then there are no other rules with any of $b_1, \ldots, b_k$ in their head. In that case a program with rules of the form (7) has the same answer sets (with respect to the definition in [36]) as the program where rules of the form (7) are replaced by the following set of rules:

$$\begin{array}{l}
b_1 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ b_2, \ldots, not\ b_k \\
b_2 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ b_1, not\ b_3, \ldots, not\ b_k \\
\ldots \\
b_k \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ b_1, \ldots, not\ b_{k-1}
\end{array}$$

3 Answer Set Planning: Using LPASS for planning

In this section we show how to do planning using LPASS – referred to as **A**nswer **S**et **P**lanning (or ASP) [26] – when the effect of actions on the world and the relationship between fluents in the world are expressed in the action description language $\mathcal{B}$. Formally, a *planning problem* with respect to $\mathcal{B}$ is specified by a triple $\langle D, \Gamma, \Delta \rangle$ where (D, Γ) is an action theory in $\mathcal{B}$ and Δ is a fluent formula (or *goal*), which a goal state must satisfy. A sequence of actions $a_1, \ldots, a_m$ is then called a *possible plan for* Δ if there exists a trajectory $s_0 a_1 s_1 \ldots a_m s_m$ in D such that s_0 and s_m satisfies Γ and Δ, respectively. Note that we define a 'possible plan' instead of a 'plan'. (In the later case the goal must be achieved on every possible trajectory.) This is because the presence of static causal laws in D allows the possibility that the effect of actions may be non-deterministic, and planning with non-deterministic actions is beyond the expressibility of LPASS. However, if D is deterministic, i.e., $|\Phi(a, s)| \leq 1$ for every pair of a state s and action a, then the notions of 'possible plan' and 'plan' coincide.

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 and *domain-independent* rules that generate action occurrences and represent the transitions between states. We now present the rules of $\Pi(D, \Gamma, \Delta)$. Our encoding closely follows the syntax of the **smodels** system as we did most of our experiments using it. We begin with the set of rules for the representation of D and Γ.

3.1 Action theory representation

We assume that actions and fluents in **A** and **F** are specified by the predicates *action*(.) and *fluent*(.), respectively, together with the necessary typed-definitions that are added for the use of variables. The encoding of Γ is straightforward and does not require any special treatment as each element in Γ can be viewed as a fact (rule without body) of Π. Since each set of literals $\{p_1, \ldots, p_n\}$ in (1)-(3) is 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, say $\{p_1, \ldots, p_n\}$, that occurs in a proposition of D a distinguished name, say n_ϕ. The constant nil denotes the set $\{\}$. A set of literals $\{p_1, \ldots, p_n\}$, with the name n_ϕ, will be encoded by the set of atoms $\{conj(n_\phi), in(p_1, n_\phi), \ldots, in(p_n, n_\phi)\}$ where $conj(n_\phi)$ specifies the type of the formula (a conjunction), and $in(f_j, n_\phi)$ indicates that f_j is a conjunct of ϕ. A proposition of the form $\mathbf{causes}(a, f, \{p_1, \ldots, p_n\})$ with $n > 0$ is encoded as a set of atoms consisting of $causes(a, f, n_\phi)$ and the set of atoms representing the formula $\phi = p_1 \wedge \ldots \wedge p_n$. Similar encodings are done for other types of propositions in D.

Example 3 To encode the block world domain in Example 1, we use the predicate $block(X)$ as the type definition for blocks. The actions and fluents are defined by the following rules:

$$
\begin{array}{rcl}
action(move(X, Y)) & \leftarrow & block(X), block(Y), X \neq Y. \\
fluent(on(X, Y)) & \leftarrow & block(X), block(Y), X \neq Y. \\
fluent(on(X, table)) & \leftarrow & block(X). \\
fluent(clear(X)) & \leftarrow & block(X).
\end{array}
$$

To encode the set of dynamic laws defined by the schema "$\mathbf{causes}(move(X, Y), on(X, Y), \{\})$ for $X \neq Y$" we write:

$$causes(move(X, Y), on(X, Y), nil) \leftarrow block(X), block(Y), X \neq Y.$$

and to encode the static law "$\mathbf{caused}(\{\neg on(Y_1, X), \ldots, \neg on(Y_n, X)\}, clear(X))$ where Y_i's are blocks different from X", we assign the name '$set\_nothing\_on(X)$' to the set $\{\neg on(Y_1, X), \ldots, \neg on(Y_n, X)\}$ and use the following rules:

$$
\begin{array}{rcl}
set(set\_nothing\_on(X)) & \leftarrow & block(X). \\
in(neg(on(Y, X)), set\_nothing\_on(X)) & \leftarrow & block(X), block(Y), X \neq Y. \\
caused(set\_nothing\_on(X), clear(X)) & \leftarrow & block(X).
\end{array}
$$

The first rule defines the set named "$set\_nothing\_on(X)$" for a block X. The second rule specifies the members of this set and the third rule encodes the static law. It is worth mentioning again that $set\_nothing\_on(X)$ is not a fluent of the domain, it is the name assigned to the set of fluent, $\{\neg on(Y_1, X), \ldots, \neg on(Y_n, X)\}$, introduced for the encoding of the static causal law. Notice also that the negative literal $\neg on(X, Y)$ is represented by the term $neg(on(X, Y))$. The encodings of the other laws are similar. □

3.2 Domain independent rules

The domain independent rules of Π are adapted mainly from [17, 13, 26, 28]. As customary in the encoding of planning problems, we assume that the length of plans we are looking for is given. We denote it by the constant *length* and use a sort *time*, whose domain is the set of integers from 0 to *length*, to represent the time moments in which the system is in. 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, and

- $occ(A,T)$: action A occurs at time T.
- $hf(F,T)$: formula F holds at time T.

In the following rules, T is a variable of the sort *time*, L and G are variables denoting *fluent literals* (written as F or $neg(F)$ for some fluent F – defined preciesly in rules (16) and (17)), S is a variable of the sort *conj* (conjunction), and A, B are variables of the sort *action*.

$$holds(L,T{+}1) \leftarrow occ(A,T), causes(A,L,S), hf(S,T). \tag{8}$$
$$holds(L,T) \leftarrow caused(S,L), hf(S,T). \tag{9}$$
$$possible(A,T) \leftarrow executable(A,S), hf(S,T). \tag{10}$$
$$holds(L,0) \leftarrow literal(L), initially(L). \tag{11}$$
$$occ(A,T) \leftarrow action(A), possible(A,T), not\ nocc(A,T). \tag{12}$$
$$nocc(A,T) \leftarrow action(A), action(B), A \neq B, occ(B,T). \tag{13}$$
$$nhf\_conj(F,T) \leftarrow conj(F), in(F_1,F), not\ hf(F_1,T). \tag{14}$$
$$hf(F,T) \leftarrow conj(F), not\ nhf\_conj(F,T). \tag{15}$$

Here, the rule (8) encodes the effects of actions and the rule (9) encodes the effects of static causal laws. The rule (10) defines a predicate that determines when an action can occur and (11) encodes the initial situation. The rules (12)-(13) generates action occurrences, one at a time[6]. The last two rules encode when a conjunction is true. The rules of inertia (or the frame axioms) and rules defining literals are encoded using the following rules:

$$literal(L) \leftarrow fluent(L). \tag{16}$$
$$literal(neg(L)) \leftarrow fluent(L). \tag{17}$$
$$contrary(F,neg(F)) \leftarrow fluent(F). \tag{18}$$
$$contrary(neg(F),F) \leftarrow fluent(F). \tag{19}$$
$$holds(L,T{+}1) \leftarrow contrary(L,G), holds(L,T), not\ holds(G,T{+}1). \tag{20}$$

The first two rules define what is considered to be a literal. The next two rules say that $neg(F)$ and F are contrary literals. The last rule says that if L holds at T and its contrary does not hold at $T+1$, then L continues to hold at $T+1$. Finally, to represent the fact that $neg(F)$ and F can not be true at the same time, the following constraint is added to Π.

$$\perp \leftarrow fluent(F), holds(F,T), holds(neg(F),T). \tag{21}$$

3.3 Goal representation

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

Definition 1 1. A fluent literal is a formula.

2. The negation of a formula is a formula.
3. A finite conjunction of formulas is a formula.

[6] These two rules can be replaced by the **smodels** cardinality constraint rule "$0\{occ(A,T) : action(A)\}1 \leftarrow time(T)$" and a set of constraint that requires that actions can occur only when they are executable and when some actions are executable then one has to occur. In many of our experiments, program with these rules yields better performance.

4. A finite disjunction of formulas is a formula.
5. If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f(X_1, \ldots, X_n)$ is a formula then $\forall X_1 : s_1, \ldots, X_n : s_n.f(X_1, \ldots, X_n)$ is a formula. When the sets $s_1, \ldots, s_n$ are clear from the context, we simply write $\forall X_1, \ldots, X_n.f(X_1, \ldots, X_n)$.
6. If $X_1, \ldots, X_n$ are variables that can have values from the sorts $s_1, \ldots, s_n$, and $f(X_1, \ldots, X_n)$ is a formula then $\exists X_1 : s_1, \ldots, X_n : s_n.f(X_1, \ldots, X_n)$ is a formula. When the sets $s_1, \ldots, s_n$ are clear from the context, we simply write $\exists X_1, \ldots, X_n.f(X_1, \ldots, X_n)$.

□

For convenience, we divide formulas into two groups: atomic and non-atomic. An *atomic formula* is a fluent literal. Other formulas are non-atomic formulas. A sort called *formula* is introduced. To encode atomic formulas, we add the following rules to Π:

$$formula(L) \quad \leftarrow \quad literal(L). \tag{22}$$

We use $conj$, $disj$, and $negation$ and $forall$ and $exists$ to represent the connectives $\wedge$, $\vee$, $\neg$ and the quantifiers $\forall$ and $\exists$, respectively. Π contains the following type definition rules:

$$\begin{aligned} formula(F) &\leftarrow conj(F). & (23)\\ formula(F) &\leftarrow disj(F). & (24)\\ formula(F) &\leftarrow negation(F, F_1). & (25)\\ formula(F) &\leftarrow forall(F). & (26)\\ formula(F) &\leftarrow exists(F). & (27) \end{aligned}$$

Each non-atomic formula ϕ will be associated with a unique name, denoted by n_ϕ, and is encoded by (possibly) a set of rules, denoted by $r(\phi)$, which is defined inductively as follows.

- For $\phi = \neg\varphi$, $r(\phi) = r(\varphi) \cup \{negation(n_\phi, n_\varphi)\}$[7].
- For $\phi = \varphi_1 \wedge \ldots \wedge \varphi_n$, $r(\phi) = \bigcup_{i=1}^{n} r(\varphi_i) \cup \{conj(n_\phi)\} \cup \{in(n_{\varphi_i}, n_\phi) \mid i = 1, \ldots, n\}$.
- For $\phi = \varphi_1 \vee \ldots \vee \varphi_n$, $r(\phi) = \bigcup_{i=1}^{n} r(\varphi_i) \cup \{disj(n_\phi)\} \cup \{in(n_{\varphi_i}, n_\phi) \mid i = 1, \ldots, n\}$.
- For $\phi = \forall X_1, \ldots, X_n.f(X_1, \ldots, X_n)$, $r(\phi)$ consists of rules defining the domains of $X_1, \ldots, X_n$, the atom $forall(n_\phi)$, and the following rule
$$in(f(X_1, \ldots, X_n), n_\phi) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n).$$
- For $\phi = \exists X_1, \ldots, X_n.f(X_1, \ldots, X_n)$, $r(\phi)$ consists of rules defining the domains of $X_1, \ldots, X_n$, the atom $exists(n_\phi)$, and the following rule
$$in(f(X_1, \ldots, X_n), n_\phi) \leftarrow in(X_1, s_1), \ldots, in(X_n, s_n).$$

For example, the conjunction $\phi = f \wedge g \wedge h$ is represented by the set of atoms $\{conj(n_\phi),\ in(f, n_\phi),\ in(g, n_\phi),\ in(h, n_\phi)\}$. We will now define $hf(F, T)$ that determines whether or not a formula F holds at the time moment T. For this purpose, we add to Π the following rules:

$$\begin{aligned} hf(F,T) &\leftarrow disj(F), in(F_1, F), hf(F_1, T). & (28)\\ hf(F,T) &\leftarrow negation(F, F_1), not\ hf(F_1, T). & (29)\\ hf(F,T) &\leftarrow literal(F), holds(F, T). & (30)\\ hf(F,T) &\leftarrow exists(F), in(F_1, F), hf(F_1, T). & (31)\\ nhf\_forall(F,T) &\leftarrow forall(F), in(F_1, F), not\ hf(F_1, T). & (32)\\ hf(F,T) &\leftarrow forall(F), not\ nhf\_forall(F, T). & (33) \end{aligned}$$

[7]To simplify notation, when φ is an atomic formula (f or $\neg f$), we have n_φ as φ.

The meanings of these rules are straightforward. The first rule says that a disjunction holds if one of its disjuncts holds. Rules (31)-(33) are for quantified formulas. Notice that rules for determining the truth value of conjunctions have been listed in the previous subsection (Rules (14) and (15)).

We now state a theorem which states that rules (14), (15)), and (28)-(33) correctly implement the evaluation of a fluent formula given the truth value of the fluents.

Theorem 1 Let S be a set of formulas, s be a state, and t be a non-negative integer. Let $\Pi = R_1 \cup R_2 \cup r(S) \cup r(s)$ where

- R_1 is the set of rules (14), (15), and (28)-(33) in which the time variable T takes the value t,
- R_2 consists of the set of rules defining literals (Rules (16) and (17)) and the set of rules defining the fluents of the domain,
- $r(s) = \{holds(l,t) \mid l \text{ is a literal and } l \in s\}$, and
- $r(S) = \bigcup_{\phi \in S} r(\phi)$.

Then,

(i) The program Π has a unique answer set, X.

(ii) For every formula ϕ in the set S, ϕ is true in s, i.e. $s \models \phi$, if and only if $hf(n_\phi, t)$ belongs to X.

Proof. See Appendix A.1 $\Box$

We now proceed towards formulating the correctness of our implementation of planning in $\mathcal{B}$.

3.4 Correctness of Π

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 (rules (8)-(33)) in which the domain of T is $\{0, \ldots, n\}$,
- the set of atoms encoding D and Γ,
- the set of atoms and rules encoding Δ, $r(\Delta)$, and
- the rule $\leftarrow$ *not* $hf(n_\Delta, n)$ that encodes the requirement that Δ holds at n.

The following result (similar to the main result in [28]) shows the equivalence between trajectories achieving Δ and answer sets of Π_n. Before stating the theorem, we introduce the following notation: for an answer set M of Π_n, we define $s_i(M) = \{f \mid f \text{ is a fluent literal and } holds(f,i) \in M\}$.

Theorem 2 For a planning problem $\langle D, \Gamma, \Delta \rangle$ with a consistent action theory (D, Γ),

(i) if $s_0 a_0 \ldots a_{n-1} s_n$ is a trajectory achieving Δ, then there exists an answer set M of Π_n such that

  1. $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$ and
  2. $s_i = s_i(M)$ for $i \in \{0, \ldots, n\}$.

  and

(ii) if M is an answer set of Π_n, then there exists an integer $0 \leq k \leq n$ such that $s_0(M)a_0 \ldots a_{k-1}s_k(k)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$. Moreover, if $k < n$ then no action is executable in the state $s_k(M)$.

Proof. See Appendix A.2 □

It is worth noticing that the second item of the theorem implies that the trajectory achieving Δ corresponds to an answer set M of Π_n could be shorter than the predefined length n. This happens when the goal is reached with a shorter sequence of actions and no action is executable in the resulting state.

The next corollary follows directly from Theorem 2.

Corollary 3.1 For a planning problem $\langle D, \Gamma, \Delta \rangle$ with a consistent and deterministic action theory (D, Γ),

1. a sequence of actions $a_0, \ldots, a_{n-1}$ is a plan achieving Δ from Γ if there exists an answer set M of Π_n such that $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$; and
2. for each answer set M of Π_n, there exists an integer $0 \leq k \leq n$ such that $a_0, \ldots, a_{k-1}$ is a plan achieving Δ from Γ where $occ(a_i, i) \in M$ for $0 \leq i < k$. Moreover, if $k < n$ then no action is executable in the state reached after executing $a_0, \ldots, a_{k-1}$ in the initial state.

4 Control Knowledge as Constraints

We now move on to the main contribution of this paper: augmenting the answer set planning (ASP) program Π in the previous section with different kinds of domain knowledge. The domain knowledge act as constraints on the answer sets of Π. For each kind of domain knowledge (also referred to as constraints) we introduce new constructs for its encoding and present a set of rules that check when a constraint is satisfied. We start with temporal domain knowledge.

4.1 Temporal Knowledge

Use of temporal domain knowledge in planning was first proposed by Bacchus and Kabanza in [1]. In their formulation temporal knowledge is used to prune the search space while planning using forward search. In their paper, temporal constraints are specified using a future linear temporal logic with a precisely defined semantics. Since their representation is separate from the action and goal representation, it is easy to add them to (or remove them from) a planning problem . 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 Definition 1 (for clarity we will henceforth refer to such formulas as *simple formulas*), or
- a goal formula of the form **goal**(φ) where φ is a formula defined in Definition 1, or
- a formula of the form **until**(φ, ψ), **always**(φ), **eventually**(φ), or **next**(φ) where φ and ψ are temporal formulas.
- (When the context is clear we will often refer to temporal formulas as formulas.)

Here, **until**, **always**, **eventually**, and **next** are temporal operators with standard meaning and **goal** is a special operator, called *goal operator*. Intuitively, a formula $\mathbf{goal}(\varphi)$ states that φ is part of the goal and must be true in a goal state. This provides a convenient way for expressing the control knowledge which depends on goal information. A temporal formula is said to be goal-independent if no goal formula occurs in it. Otherwise, it is goal-dependent. Bacchuss and Kabanza [1] observed that useful temporal knowledge in planning is often goal-dependent. In the block world domain, the following goal-dependent formula[8]:

$$\mathbf{always}(\mathbf{goal}(on(X, table)) \wedge on(X, table) \supset \mathbf{next}(on(X, table))) \tag{34}$$

can be used to express that if the goal is to have a block on the table and it is already on the table then it should be still on the table in the next moment of time. This has the effect of preventing the agent from superfluously picking up a block from the table if it is supposed to be in the table in a goal state.

It is worth noting that under this definition, temporal operators can be nested many times but the goal operator **goal** cannot be nested. For instance, if φ is a fluent formula, $\mathbf{always}(\mathbf{next}(\varphi))$ is a temporal formula, but $\mathbf{goal}(\mathbf{goal}(\varphi))$ is not.

Temporal formulas which do not contain the **goal** operator (i.e. goal-independent formulas) will be interpreted over a infinite sequence of states of D, denoted by $I = \langle s_0, s_1, \ldots, \rangle$. On the other hand temporal formulas which contain **goal** (i.e. goal-dependent formulas) will be evaluated with respect to a pair $\langle I, \varphi\rangle$ where I is a sequence of states and φ is a simple formula. We now formally define them using two separate definitions. Definition 2 deals with goal-independent formulas while Definition 3 is concerned with general temporal formulas (possibly goal-dependent).

Definition 2 (See [1]) Let $I = \langle s_0, s_1, \ldots, s_n, \ldots\rangle$ be a sequence of states of D. Let f_1 and f_2 be goal-independent temporal formulas, t be a non-negative integer, and f_3 be a simple formula. Let $I_t = \langle s_t, s_{t+1}, \ldots, \rangle$ denoting the subsequence of I starting from s_t.

I entails or satisfies a goal-independent temporal formula f, denoted by $I \models f$, if $I_0 \models f$ where

- $I_t \models f_3$ iff $s_t \models f_3$.
- $I_t \models \mathbf{until}(f_1, f_2)$ iff there exists $t \le t_2$ such that $I_{t_2} \models f_2$ and for all $t \le t_1 < t_2$ we have $I_{t_1} \models f_1$.
- $I_t \models \mathbf{next}(f_1)$ iff $I_{t+1} \models f_1$.
- $I_t \models \mathbf{eventually}(f_1)$ iff there exists $t \le t_1$ such that $I_{t_1} \models f_1$.
- $I_t \models \mathbf{always}(f_1)$ iff for all $t \le t_1$ we have $I_{t_1} \models f_1$.

For a finite sequence of states $I = \langle s_0, \ldots, s_n\rangle$ and a goal-independent temporal formula f, we say I entails (or satisfies) f, denoted by $I \models f$, if $I' \models f$ where $I' = \langle s_0, \ldots, s_n, s_n, \ldots\rangle$. □

Next we define when temporal formulas are entailed or satisfied by a sequence of states and a goal.

Definition 3 Let $I = \langle s_0, s_1, \ldots, s_n, \ldots\rangle$ be a sequence of states of D and φ be a simple formula denoting the goal. Let f_1 and f_2 be temporal formulas (possibly goal dependent), t be a non-negative integer, and f_3 be a simple formula. Let $I_t = \langle s_t, s_{t+1}, \ldots, \rangle$.

I entails or satisfies a temporal formula f with respect to φ, denoted by $\langle I, \varphi\rangle \models f$, if $\langle I_0, \varphi\rangle \models f$ where

- $\langle I_t, \varphi\rangle \models f_3$ iff $s_t \models f_3$.

[8]As before we use the convention that a formula with variables represents the set of its ground instantiations.

- $\langle I_t, \varphi \rangle \models \mathbf{goal}(f_3)$ iff $\varphi \models f_3$
- $\langle I_t, \varphi \rangle \models \mathbf{until}(f_1, f_2)$ iff there exists $t \le t_2$ such that $\langle I_{t_2}, \varphi \rangle \models f_2$ and for all $t \le t_1 < t_2$ we have $\langle I_{t_1}, \varphi \rangle \models f_1$.
- $\langle I_t, \varphi \rangle \models \mathbf{next}(f_1)$ iff $\langle I_{t+1}, \varphi \rangle \models f_1$.
- $\langle I_t, \varphi \rangle \models \mathbf{eventually}(f_1)$ iff there exists $t \le t_1$ such that $\langle I_{t_1}, \varphi \rangle \models f_1$.
- $\langle I_t, \varphi \rangle \models \mathbf{always}(f_1)$ iff for all $t \le t_1$ we have $\langle I_{t_1}, \varphi \rangle \models f_1$.

For a finite sequence of states $I = \langle s_0, \ldots, s_n \rangle$, a temporal formula f, and a simple formula φ we say I satisfies f with respect to φ, denoted by $\langle I, \varphi \rangle \models f$, if $\langle I', \varphi \rangle \models f$ where $I' = \langle s_0, \ldots, s_n, s_n, \ldots \rangle$. $\square$

Similar to simple non-atomic formulas, temporal formulas can be encoded in ASP using constants, atoms, and rules. We do this in two steps.

First, each temporal formula, say ϕ, is represented by a set of rules $r(\phi)$, which is defined inductively as follows.

- For simple formulas ϕ, $r(\phi)$ is defined as before (Section 3.3).
- For $\phi = \mathbf{always}(\varphi)$, $r(\phi) = r(\varphi) \cup \{always(n_\phi, n_\varphi) \leftarrow\}$.
- For $\phi = \mathbf{next}(\varphi)$, $r(\phi) = r(\varphi) \cup \{next(n_\phi, n_\varphi) \leftarrow\}$.
- For $\phi = \mathbf{eventually}(\varphi)$, $r(\phi) = r(\varphi) \cup \{eventually(n_\phi, n_\varphi) \leftarrow\}$.
- For $\phi = \mathbf{until}(\varphi, \psi)$, $r(\phi) = r(\varphi) \cup r(\psi) \cup \{until(n_\phi, n_\varphi, n_\psi) \leftarrow\}$.
- For $\phi = \mathbf{goal}(\psi)$, $r(\phi) = r(\psi)$ with n_ϕ is the name associated to $\mathbf{goal}(\psi)$; in what follows, we will use $n_{goal(\psi)}$ to denote the name assigned to the goal formula $\mathbf{goal}(\psi)$.

For example, $r(until(f, next(g)))$ is the set of facts $\{until(f_1, f, f_2), next(f_2, g)\}$, where f_1 and f_2 are the names assigned to $\mathbf{until}(f, \mathbf{next}(g))$ and $\mathbf{next}(g)$, respectively. When encoding temporal formulas with variables we can instantiate them first and then encode the instantiation, or do the encoding as illustrated by the following example, where we encode the temporal formula (34).

Example 4 To encode the temporal formula (34), we name the sub-formulas as follows, where the variables in the names play the same role as the variables in the formulas.

1. $n_1(X) = \mathbf{next}(on(X, table))$
2. $n_2(X) = \mathbf{goal}(on(X, table)) \wedge on(X, table)$
3. $n_3(X) = \neg(\mathbf{goal}(on(X, table)) \wedge on(X, table)) \vee \mathbf{next}(on(X, table))$
4. $n_4(X) = \mathbf{always}(\mathbf{goal}(on(X, table)) \wedge on(X, table) \supset \mathbf{next}(on(X, table)))$

Then the encoding of (34) is the encoding of $n_4(X)$.

The rules encoding $n_1(X)$ are:

$$next(n_1(X), on(X, table)) \quad \leftarrow \quad block(X).$$

$n_2(X)$ is encoded by the following rules:

$$\begin{array}{lcl} conj(n_2(X)) & \leftarrow & block(X). \\ in(n_{goal(on(X,table))}, n_2(X)) & \leftarrow & block(X). \\ in(on(X, table), n_2(X)) & \leftarrow & block(X). \end{array}$$

The set of rules encoding $n_3(X)$ contains the rules for encoding $n_2(X)$ and $n_1(X)$ and the following rules:

$$\begin{array}{lcl} disj(n_3(X)) & \leftarrow & block(X). \\ in(n_1(X), n_3(X)) & \leftarrow & block(X). \\ in(negation(n_2(X)), n_3(X)) & \leftarrow & block(X). \end{array}$$

Finally, the set of rules encoding $n_4(X)$ contains $r(n_3(X))$ and the following rule:

$$always(n_4(X), n_3(X)) \quad \leftarrow \quad block(X).$$

□

To complete the encoding of temporal constraints, we now provide the rules for evaluating temporal formulas. To achieve that we extend the earlier set of rules in Section 3.3 that define the predicate hf. As when defining entailment of temporal formulas, we first consider goal-independent temporal formulas. The rules needed for evaluating temporal formulas whose first level operator is different than the **goal** operator are as follows:

$$hf(N,T) \leftarrow until(N,N_1,N_2), hf\_during(N_1,T,T'), hf(N_2,T'). \tag{35}$$
$$hf(N,T) \leftarrow always(N,N_1), hf\_during(N_1,T,n+1). \tag{36}$$
$$hf(N,T) \leftarrow eventually(N,N_1), hf(N_1,T'), T \leq T'. \tag{37}$$
$$hf(N,T) \leftarrow next(N,N_1), hf(N_1,T+1). \tag{38}$$
$$nhf\_during(N,T,T') \leftarrow not\ hf(N,T''), T \leq T'' < T'. \tag{39}$$
$$hf\_during(N,T,T') \leftarrow hf(N,T), not\ nhf\_during(N,T,T'). \tag{40}$$

In the above rules, for a temporal formula N, $hf(N,T)$ means that N is satisfied by $\langle s_T, s_{T+1}, \ldots s_n \rangle$, where s_T refers to the state corresponding to time point T. With this meaning the rules encode Definition 2 in a straightforward way.

The next theorem shows that rules (35)-(40) correctly implement the semantics of goal-independent temporal formulas.

Theorem 3 Let S be a set of goal-independent temporal formulas, $I = \langle s_0, s_1 \ldots s_n \rangle$ be a sequence of states, and I_t denote $\langle s_t, \ldots s_n \rangle$. Let

$$\Pi = R_1 \cup R_2 \cup r(S) \cup r(I)$$

where

- R_1 consists of the set of rules (14), (15), and (28)-(33) in which the domain of T is $\{0, \ldots, n\}$, the set of rules (16)-(17)), and the set of rules defining the fluents of the domain,
- R_2 is the set of rules (35)-(40) in which the domain of T is $\{0, \ldots, n\}$,
- $r(I) = \cup_{t=0}^{n} \{holds(l,t) \mid l \text{ is a fluent literal and } l \in s_t\}$, and
- $r(S) = \bigcup_{\phi \in S} r(\phi)$.

Then,

(i) The program Π has a unique answer set, X.

(ii) For every temporal formula ϕ in the set S, ϕ is true in I_t, i.e., $I_t \models \phi$, if and only if $hf(n_\phi, t)$ belongs to X.

Proof. See Appendix A.1 $\Box$

Having defined temporal constraints and specified when they are satisfied, adding temporal knowledge to a planning problem in ASP is easy. We must encode the knowledge as a temporal formula[9] and then add the set of rules representing this formula and the rules (35)-(40) to Π. Finally, we need to add the constraint that requires that the goal is true at the final state and the temporal formula is satisfied. More precisely, for a planning problem $\langle D, \Gamma, \Delta \rangle$ and a goal-independent temporal formula ϕ, let Π_n^{TLP} be the program consisting of

- the program Π_n,
- the rules (35)-(40)
- the rules encoding ϕ and the constraint $\leftarrow$ *not* $hf(n_\phi, 0)$.

The next theorem is about the correctness of Π_n^{TLP}.

Theorem 4 For a planning problem $\langle D, \Gamma, \Delta \rangle$ with a consistent action theory (D, Γ) and a goal-independent temporal formula ϕ,

(i) if $s_0 a_0 \ldots a_{n-1} s_n$ is a trajectory achieving Δ and $I \models \phi$ where $I = \langle s_0, \ldots, s_n \rangle$, then there exists an answer set M of Π_n^{TLP} such that

1. $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$,
2. $s_i = s_i(M)$ for $i \in \{0, \ldots, n\}$, and
3. $hf(n_\phi, 0) \in M$.

and

(ii) if M is an answer set of Π_n^{TLP}, then there exists an integer $0 \leq k \leq n$ such that

1. $s_0(M) a_0 \ldots a_{k-1} s_k(M)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$ and
2. $I \models \phi$ where $I = \langle s_0(M), \ldots, s_n(M) \rangle$.

Proof. Follows from Theorem 2 and Theorem 3. $\Box$

The above theorem shows how control knowledge represented as goal-independent temporal formulas can be exploited in ASP. We will now extend this result to allow control knowledge expressed using goal-dependent temporal formulas. Based on Definition 3, where entailment of goal-dependent temporal formulas is defined, we will need to encode $\varphi \models f_3$. To simplify this encoding we make the same assumption that is made in most classical planning literature including [1]: our goals will be a conjunction of literals. I.e., the goal Δ in a planning problem $\langle D, \Gamma, \Delta \rangle$ will be a set of literals and each goal formula occurring in a temporal formula representing our control knowledge is of the form $\mathbf{goal}(F)$ where F is a fluent literal. In the rest of this section, whenever we refer to a planning problem or a goal-dependent temporal formula we assume that they satisfy this assumption. Let $\langle D, \Gamma, \Delta \rangle$ be a planning problem and ϕ be a temporal formula. $\Pi_n^{TLP+Goal}$ be the program consisting of Π_n^{TLP} and the rule

$$hf(n_{goal(L)}, T) \quad \leftarrow \quad literal(L), conj(n_\Delta), in(L, n_\Delta). \tag{41}$$

Notice that because Δ is a conjunction of literals, rule (41) is sufficient for determining whether $\Delta \models l$ for some fluent literal l holds or not. The next theorem is about the correctness of $\Pi_n^{TLP+Goal}$.

[9] A set of temporal formulas can be viewed as a conjunction of temporal formulas.

Theorem 5 For a planning problem $\langle D, \Gamma, \Delta\rangle$ with a consistent action theory (D, Γ) and a temporal formula ϕ,

(i) if $s_0 a_0 \ldots a_{n-1} s_n$ is a trajectory achieving Δ and $\langle I, \Delta\rangle \models \phi$ where $I = \langle s_0, \ldots, s_n\rangle$, then there exists an answer set M of $\Pi_n^{TLP+Goal}$ such that

1. $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$,
2. $s_i = s_i(M)$ for $i \in \{0, \ldots, n\}$, and
3. $hf(n_\phi, 0) \in M$.

and

(ii) if M is an answer set of $\Pi_n^{TLP+Goal}$, then there exists an integer $0 \leq k \leq n$ such that

1. $s_0(M) a_0 \ldots a_{k-1} s_k(M)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$ and
2. $\langle I, \Delta\rangle \models \phi$ where $I = \langle s_0(M), \ldots, s_n(M)\rangle$.

Proof. To prove this theorem, we first need to modify Theorem 3 by (i) allowing goal-dependent formulas to be in the set S; (ii) adding a goal Δ and the rule (41) to the program Π of Theorem 3. The proof of this modified theorem is very similar to the proof of Theorem 3. This result, together with Theorem 2, proves the conclusion of this theorem. $\Box$

4.2 Procedural Knowledge

Procedural knowledge can be thought of as an under-specified sketch of the plans to be generated. The language constructs of procedural knowledge that we use in this paper are inspired by GOLOG, an Algol-like logic programming language for agent programming, control and execution; and based on a situation calculus theory of actions [25]. GOLOG has been primarily used as a programming language for high-level agent control in dynamical environments (see e.g. [8]). Although a planner can be written as a GOLOG program (See Chapter 10 of [38]), our view of a GOLOG program in this paper is different. We view it as an incompletely specified plan (or a form of procedural knowledge) that includes non-deterministic choice points that are filled in by the planner. For example, the procedural knowledge (which is very similar to a GOLOG program) $a_1; a_2; (a_3|a_4|a_5); f$ 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. A planner, when given this procedural knowledge needs only to decide which one of a_3, a_4, or a_5 it should choose as its third action.

We now formally define the syntax of our procedural knowledge, which – keeping with the GOLOG terminology – we refer to as a program.

Definition 4 (Program) For an action theory (D, Γ),

1. an action a is a program;
2. a simple formula ϕ (as defined in Subsection 3.3) is a program[10];
3. if p_i's are programs then $p_1; \ldots; p_n$ is a program, and $p_1, \ldots, p_n$ are said to occur in it;
4. if p_i's are programs then $p_1 \mid \ldots \mid p_n$ is a program, and $p_1, \ldots, p_n$ are said to occur in it;
5. if p_1 and p_2 are programs and ϕ is a simple formula then "**if** ϕ **then** p_1 **else** p_2" is a program, and $p1$ and p_2 are said to occur in it;

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

6. if p is a program and ϕ is a simple formula then "**while** ϕ **do** p" is a program, and p is said to occur in it; and

7. if $X_1, \ldots, X_n$ are variables of sort $s_1, \ldots, s_n$, respectively, $p(X_1, \ldots, X_n)$ is a program, and $f(X_1, \ldots, X_n)$ is a simple formula, then **pick**$(X_1, \ldots, X_n, f(X_1, \ldots, X_n), p(X_1, \ldots, X_n))$ is a program, and $p(X_1, \ldots, X_n)$ is said to occur in it.

□

In general programs that are used in the construction of other programs are said to occur in them. Programs defined in Item 1 and 2 are called primitive; and others are referred to as non-primitive. For a program p, by $progs(p)$ we denote the set of programs that occur in p. More precisely, $progs(p) = \{p\}$ if p is a primitive program; and $progs(p) = \bigcup_{p_i \text{occurs in } p} progs(p_i)$ if p is a non-primitive program. Notice that the definition of programs allows "recursive" programs like "**while** ϕ_1 **do** q" and "**while** ϕ_2 **do** p" where p and q refer to the first and second program, respectively. It is easy to see that there are situations in which the execution of a program p or q may never stop. In this paper, we are not interested in such programs as our purpose is to use programs to construct finite plans. Towards that purpose, we define a notion of a well-defined set of programs as follows.

A program p depends on a program q, denoted by $q < p$, if $q \in progs(p)$. Let $<^*$ be the transitive closure of $<$. Given a set S of programs of an action theory (D, Γ), the $<$ relation will induce a partial order on S. We say that S is *well-defined* if (i) for every non-primitive program $p \in S$, $progs(p)$ contains at least one action and (ii) there exists no program p such that $p <^* p$ holds. It is worth noting that recursive programs in conventional sense (i.e., with break conditions) are normally well-defined. For instance, if $p(n)$ is a program with an integer parameter n then the set of program occurring in $p(n)$, defined by "**while** $n > 0$ **do** $p(n-1)$", is a well-defined program. For this reason, we will limit ourselves to the study of well-defined sets of programs. *From now on, whenever we say a program p, we assume p to be such that it guarantees $progs(p)$ to be a well-defined set of programs.* We illustrate the above definition with the following example.

Example 5 In this example, we introduce the elevator domain from [25] which we use in our initial experiments (Section 4.4). The fluents in this domain and their intuitive meaning are as follows:

- $on(N)$ - the request service light of the floor N is on, indicating a service is requested at the floor N,
- *opened* - the door of the elevator is open, and
- $currentFloor(N)$ - the elevator is currently at the floor N.

The actions in this domain and their intuitive meaning are as follows:

- $up(N)$ - move up to floor N,
- $down(N)$ - move down to floor N,
- $turnoff(N)$ - turn off the indicator light of the floor N,
- *open* - open the elevator door, and
- *close* - close the elevator door.

The domain description is as follows:

$$D_{elevator} = \begin{cases} \textbf{causes}(up(N), currentFloor(N), \{\}) \\ \textbf{causes}(down(N), currentFloor(N), \{\}) \\ \textbf{causes}(turnoff(N), \neg on(N), \{\}) \\ \textbf{causes}(open, opened, \{\}) \\ \textbf{causes}(close, \neg opened, \{\}) \\ \textbf{caused}(\{currentFloor(M)\}, \neg currentFloor(N)) \text{ for all } M \neq N \\ \textbf{executable}(up(N), \{currentFloor(M), \neg opened\}) \text{ for all } M < N \\ \textbf{executable}(down(N), \{currentFloor(M), \neg opened\}) \text{ for all } M > N \\ \textbf{executable}(turnoff(N), \{currentFloor(N)\}) \\ \textbf{executable}(open, \{\}) \\ \textbf{executable}(close, \{\}) \\ \textbf{executable}(null, \{\}) \end{cases}$$

We consider arbitrary initial states where *opened* is false, $currentFloor(N)$ is true for a particular N and a set of $on(N)$ is true; and our goal is to have $\neg on(N)$ for all N. In planning to achieve such a goal, we can use the following set of procedural domain knowledge (or programs). Alternatively, in the terminology of GOLOG, we can say that the following set of programs can be used to control the elevator, so as to satisfy service requests – indicated by the light being on – at different floors.

$$\begin{aligned} go\_floor(N) &\equiv currentFloor(N) | up(N) | down(N). \\ serve(N) &\equiv go\_floor(N); turnoff(N); open; close. \\ serve\_a\_floor &\equiv \textbf{pick}(N, on(N), serve(N)). \\ park &\equiv \textbf{if } currentFloor(0) \textbf{ then } open \textbf{ else } [down(0); open]. \\ control &\equiv [\textbf{while } \exists X.[on(X)] \textbf{ do } serve\_a\_floor]; park \end{aligned}$$

Observe that $go\_floor(N)$ is a choice of actions (Item 4, Definition 4); $serve(N)$ is a sequence of programs (Item 3); $serve\_a\_floor$ is a choice of arguments (Item 7); $park$ is an example of the **if** ... **then** construct; and *control* is a **while** loop. □

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*. Intuitively, α is a trace of a program p means that $a_0, \ldots, a_{n-1}$ is a sequence of actions (and α is a corresponding trajectory) that is consistent with the sketch provided by the procedural constraint p starting from the initial state s_0. Alternatively, it can be thought of as the program p *unfolds* to the sequence of actions $a_0, \ldots, a_{n-1}$ in state s_0. We now formally define the notion of a *trace*.

Definition 5 (Trace) Let p be a program. We say that a trajectory $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trace of p if one of the following conditions is satisfied:

- $p = a$ and a is an action, $n = 1$ and $a_0 = a$;
- $p = \phi$, $n = 0$ and ϕ holds in s_0;
- $p = p_1; p_2$, and 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;
- $p = p_1 \mid \ldots \mid p_n$, and $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p_i for some $i \in \{1, \ldots, n\}$;
- $p = \textbf{if } \phi \textbf{ then } p_1 \textbf{ else } p_2$, and $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p_1 if ϕ holds in s_0 or $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p_2 if $\neg\phi$ holds in s_0;
- $p = \textbf{while } \phi \textbf{ do } p_1$, $n = 0$ and $\neg\phi$ holds in s_0, or
  ϕ holds in s_0 and there exists some $i > 0$ 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; or

- $p =$ **pick**$(\vec{X}, f(\vec{X}), q(\vec{X}))$, and there exists a constant $\vec{x}$ of the sort of $\vec{X}$ such that $f(\vec{x})$ holds in s_0 and $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trace of $q(\vec{x})$.

$\square$

Similar to our earlier encoding of formulas, we will assign to each program a name (with the exception of actions and formulas), provide rules for the construction of programs, and use the prefix notation. Again, for a program p, let n_p denote the name assigned to p. The set of rules representing a program p, which is not an action or a formula, will be denoted by $r(p)$ and is defined inductively as follows.

1. For $p = p_1; p_2$, $r(p) = \{proc(n_p, n_{p_1}, n_{p_2})\} \cup r(p_1) \cup r(p_2)$.
2. For $p = p_1 \mid \ldots \mid p_n$, $r(p) = \bigcup_{i=1}^{n} r(p_i) \cup \{in(n_{p_i}, n_p) \mid 1 \leq i \leq n\} \cup \{choiceAction(n_p)\}$.
3. For $p =$ **if** ϕ **then** p_1 **else** p_2, $r(p) = r(\phi) \cup r(p_1) \cup r(p_2) \cup \{if(n_p, n_\phi, n_{p_1}, n_{p_2})\}$.
4. For $p =$ **while** ϕ **do** p_1, $r(p) = r(\phi) \cup r(p_1) \cup \{while(n_p, n_\phi, n_{p_1})\}$.
5. For $p =$ **pick**$(\vec{x}, f(\vec{x}), p_1(\vec{x}))$, $r(p) = r(p_1(\vec{x})) \cup r(f(\vec{x})) \cup \{choiceArgs(n_p, n_{f(\vec{x})}, n_{p_1(\vec{x})})\}$.

Example 6 In this example we present the encoding of the programs from Example 5.

We start with the set of rules encoding the program $go\_floor(N)$:

$$\begin{array}{rl} choiceAction(go\_floor(N)) & \leftarrow floor(N). \\ in(currentFloor(N), go\_floor(N)) & \leftarrow floor(N). \\ in(up(N), go\_floor(N)) & \leftarrow floor(N). \\ in(down(N), go\_floor(N)) & \leftarrow floor(N). \end{array}$$

The following rules encode the program $serve(N)$:

$$\begin{array}{rl} proc(serve(N), go\_floor(N), serve\_tail\_1(N)) & \leftarrow floor(N). \\ proc(serve\_tail\_1(N), turnoff(N), open\_close) & \leftarrow floor(N). \\ proc(open\_close, open, close) & \leftarrow \end{array}$$

To encode the program $serve\_a\_floor$, we need the following rule:

$$choiceArgs(serve\_a\_floor, on(N), serve(N)) \leftarrow floor(N).$$

The following rules encode the program $park$:

$$\begin{array}{rl} if(park, currentFloor(0), open, park\_1) & \leftarrow \\ proc(park\_1, down(0), open) & \leftarrow \end{array}$$

Finally, the encoding of program $control$ consists of the following rules:

$$\begin{array}{rl} proc(control, while\_service\_needed, park) & \leftarrow \\ while(while\_service\_needed, existOn, serve\_a\_floor) & \leftarrow \\ exists(existOn) & \leftarrow \\ in(existOn, on(N)) & \leftarrow floor(N). \end{array}$$

$\square$

We now present the LPASS rules that realize the operational semantics of programs. Intuitively, $trans(p, t_1, t_2)$ is true in an answer set M iff $s_{t_1}(M)a_{t_1} \dots a_{t_2-1}s_{t_2}(M)$ is a trace of the program p[11].

$$
\begin{array}{rcll}
trans(A, T, T+1) & \leftarrow & action(A), occ(A, T). & (42)\\
trans(F, T_1, T_1) & \leftarrow & formula(F), hf(F, T_1). & (43)\\
trans(P, T_1, T_2) & \leftarrow & proc(P, P_1, P_2), T1 \leq T' \leq T_2, & (44)\\
 & & trans(P_1, T_1, T'), trans(P_2, T', T_2). & (45)\\
trans(N, T_1, T_2) & \leftarrow & choiceAction(N), & (46)\\
 & & in(P_1, N), trans(P_1, T_1, T_2). & \\
trans(I, T_1, T_2) & \leftarrow & if(I, F, P_1, P_2), hf(F, T_1), trans(P_1, T_1, T_2). & (47)\\
trans(I, T_1, T_2) & \leftarrow & if(I, F, P_1, P_2), not\ hf(F, T_1), trans(P_2, T_1, T_2). & (48)\\
trans(W, T_1, T_2) & \leftarrow & while(W, F, P), hf(F, T_1), T_1 < T' \leq T_2, & (49)\\
 & & trans(P, T_1, T'), trans(W, T', T_2). & \\
trans(W, T, T) & \leftarrow & while(W, F, P), not\ hf(F, T). & (50)\\
trans(S, T_1, T_2) & \leftarrow & choiceArgs(S, F, P), hf(F, T_1), trans(P, T_1, T_2). & (51)\\
trans(\mathbf{null}, T, T) & \leftarrow & & (52)
\end{array}
$$

Here **null** denotes a dummy program that performs no action. This action is added to allow programs of the form **if** φ **then** p to be considered (this will be represented as **if** φ **then** p **else** **null**). The rules are used for determining whether a trajectory – encoded by answer sets of the program Π_n – is a trace of a program or not. As with temporal constraints, this is done inductively over the structure of programs. The rules (42) and (43) are for programs consisting of an action and a simple formula respectively. The other rules are for the remaining cases. For instance, the rule (49) states that the trajectory from T_1 to T_2 is a trace of a while loop "**while** F **do** P", named W and encoded by the atom $while(W, F, P)$, if the formula F holds at T_1 and there exists some T', $T_1 < T' \leq T_2$ such that the trajectory from T_1 to T' is a trace of P and the trajectory from T' to T_2 is a trace of W; and the rule (50) states that the trajectory from T to T is a trace of W if the formula F does not holds at T. These two rules effectively determine whether the trajectory from T_1 to T_2 is a trace of $while(W, F, P)$. The meanings of the other rules are similar.

To specify that a plan of length n starting from an initial state must obey the sketch specified by a program p, all we need to do is to have a rule $\leftarrow not\ trans(n_p, 0, n)$. We now formulate the correctness of our above encoding of procedural knowledge given as programs, and relate the traces of program with the answer sets of its LPASS encoding. Let Π_n^T be the program obtained from Π_n by (i) adding the rules (42)-(52), (ii) adding $r(p)$, and (iii) replacing the goal constraint with $\leftarrow not\ trans(n_p, 0, n)$. The following theorem is similar to Theorem 2.

Theorem 6 Let (D, Γ) be a consistent action theory and p be a program. Then,

(i) for every answer set M of Π_n^T with $occ(a_i, i) \in M$ for $i \in \{0, \dots, n-1\}$, $s_0(M)a_0 \dots a_{n-1}s_n(M)$ 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 an answer set M of Π_n^T such that $s_j = s_j(M)$ and $occ(a_i, i) \in M$ for $j \in \{0, \dots, n\}$ and $i \in \{0, \dots, n-1\}$.

Proof. See Appendix A.3 □

Now, to do planning using procedural constraints all we need to do is to add the goal constraint to Π_n^T, which will filter out all answer sets where the goal is not satisfied in time point n, and at the same time will use the sketch provided by the program p.

[11]Recall that we define $s_i(M) = \{holds(f, i) \in M \mid f \text{ is a fluent literal}\}$.

4.3 HTN Knowledge

The programs in the previous section are good for representing procedural knowledge but prove cumbersome for encoding partial ordering information. For example, to represent that any sequence containing the n programs $p_1, \ldots, p_n$, 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-determinism construct. For $n = 3$, the program fragment would be $(p_1; p_2; p_3 | p_1; p_3; p_2 | p_3; p_1; p_2)$. Alternatively, the use of the *concurrent construct* $\|$ from [12], where $p \| q$ represents the set consisting of two programs $p; q$ and $q; p$, is not very helpful either. This deficiency of pure procedural constructs of the type discussed in the previous section prompted us to look at the constructs in HTN planning [39]. The partial ordering information allowed in HTN descriptions serves the purpose. Thus all we need is to have the constraint that says p_1 must occur before p_2.

The constructs in HTN by themselves are not expressive enough either as they do not have procedural constructs such as procedures, conditionals, or loops, and expressing a while loop using pure HTN constructs is not trivial. Thus we decided to combine the HTN and procedural constructs and go further than the initial attempt in [5] where complex programs are not allowed to occur within HTN programs.

We now define a more general notion of program that allows both procedural and HTN constructs. For that we need the following notions. Let $S = \{p_1, \ldots, p_k\}$ be a set of programs. Assume that n_i, $1 \leq i \leq k$, is the name assigned to the program p_i.

- An ordering constraint over S has the form $n_i \prec n_j$ where $n_i \neq n_j$. Intuitively, an ordering constraint $n_i \prec n_j$ requires that the program p_i has to be executed before the program p_j.
- A truth constraint is of the form (n_i, ϕ), (ϕ, n_i), or (n_i, ϕ, n_t), where ϕ is a formula.

  A truth constraint of the form (n_i, ϕ) (resp. (ϕ, n_i)) requires that immediately after (resp. immediately before) the execution of p_i, ϕ must hold.

  On the other hand, a constraint of the form (n_i, ϕ, n_t) indicates that ϕ must hold immediately after the time p_i is executed until p_t begins its execution. For this reason, we will assume that whenever (n_i, ϕ, n_t) belongs to C, so does $n_i \prec n_t$.

Definition 6 (General programs) For an action theory (D, Γ),

- an action a is a general program;
- a simple formula ϕ (as defined in Subsection 3.3) is a general program;
- if p_i's are general programs then $p_1; \ldots; p_n$ is a general program, and $p_1, \ldots, p_n$ are said to occur in it;
- if p_i's are general programs then $p_1 \mid \ldots \mid p_n$ is a general program, and $p_1, \ldots, p_n$ are said to occur in it;
- if p_1 and p_2 are general programs and ϕ is a simple formula then "**if** ϕ **then** p_1 **else** p_2" is a general program, and p_1 and p_2 are said to occur in it;
- if p is a general program and ϕ is a simple formula then "**while** ϕ **do** p" is a general program, and p is said to occur in it;
- if $X_1, \ldots, X_n$ are variables of sort $s_1, \ldots, s_n$, respectively, $p(X_1, \ldots, X_n)$ is a general program, and $f(X_1, \ldots, X_n)$ is a simple formula, then **pick**$(X_1, \ldots, X_n, f(X_1, \ldots, X_n), p(X_1, \ldots, X_n))$ is a general program, and $p(X_1, \ldots, X_n)$ is said to occur in it; and
- if S is a set of general programs and C is a set of ordering or truth constraints then the pair (S, C) is a general program, and the programs in S are said to occur in (S, C).

□

The notion of well-definedness of a set of general programs is defined similar to the notion of well-defined sets of programs and as before we assume that we only consider general programs p such that $progs(p)$ is a well-defined set of general programs.

As in the case of programs, the operational semantics of general programs is defined using the notion of trace. We now define the notion of a trace of general programs.

Definition 7 (Trace of general programs) Let p be a general program. We say that a trajectory $s_0a_0\ldots a_{n-1}s_n$ is a trace of p if one of the following conditions is satisfied:

1. $p = a$ and a is an action, $n = 1$ and $a_0 = a$;
2. $p = \phi$, $n = 0$ and ϕ holds in s_0;
3. $p = p_1; p_2$, and there exists an i such that $s_0a_0 \ldots s_i$ is a trace of p_1 and $s_ia_i \ldots s_n$ is a trace of p_2;
4. $p = p_1 \mid \ldots \mid p_n$, and $s_0a_0 \ldots a_{n-1}s_n$ is a trace of p_i for some $i \in \{1, \ldots, n\}$;
5. $p =$ **if** ϕ **then** p_1 **else** p_2, and $s_0a_0 \ldots a_{n-1}s_n$ is a trace of p_1 if ϕ holds in s_0 or $s_0a_0 \ldots a_{n-1}s_n$ is a trace of p_2 if $\neg\phi$ holds in s_0;
6. $p =$ **while** ϕ **do** p_1, $n = 0$ and $\neg\phi$ holds in s_0, or ϕ holds in s_0 and there exists some $i > 0$ such that $s_0a_0 \ldots s_i$ is a trace of p_1 and $s_ia_i \ldots s_n$ is a trace of p;
7. $p =$ **pick**$(\vec{X}, f(\vec{X}), q(\vec{X}))$, and there exists a constant $\vec{x}$ of the sort of $\vec{X}$ such that $f(\vec{x})$ holds in s_0 and $s_0a_0 \ldots a_{n-1}s_n$ is a trace of $q(\vec{x})$; or
8. $p = (S, C)$ and there exists $j_0{=}0 \leq j_1 \leq \ldots \leq j_k{=}n$ 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:
   (a) for each l, $1 \leq l \leq k$, α_l is a trace of p_{i_l},
   (b) if $n_t < n_l \in C$ then $i_t < i_l$,
   (c) 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
   (d) if $(n_t, \phi, n_l) \in C$ then ϕ holds in $s_{j_t}, \ldots, s_{j_{l-1}}$.

□

The last item of the above definition can be visualized by the following illustration:

$$\underbrace{s_0a_0s_1\ldots a_{j_1-1}s_{j_1}}_{\substack{\alpha_1\\ \uparrow\\ \downarrow\\ \text{trace of } p_{i_1}}}\underbrace{s_{j_1}a_{j_1}\ldots a_{j_2-1}s_{j_2}}_{\substack{\alpha_2\\ \uparrow\\ \downarrow\\ \text{trace of } p_{i_2}}}\ldots\underbrace{s_{j_{l-1}}a_{j_{l-1}+1}\ldots a_{j_l-1}s_{j_l}}_{\substack{\alpha_l\\ \uparrow\\ \downarrow\\ \text{trace of } p_{i_l}}}\ldots\underbrace{s_{j_{k-1}}a_{j_{k-1}+1}\ldots a_{j_k-1}s_{j_k}}_{\substack{\alpha_k\\ \uparrow\\ \downarrow\\ \text{trace of } p_{i_k}}}$$

Next we show how to represent general programs. Similar to programs in the previous section, we will assign names to general programs and their elements. A general program $p = (S, C)$ is encoded by the set of atoms and rules

$$r(p) = \{htn(n_p, n_S, n_C)\} \cup r(S) \cup r(C)$$

where $r(S)$ and $r(C)$ is the set of atoms and rules encoding S and C and is described below. Recall that S is a set of programs and C is a set of constrains. Both S and C are assigned unique names, n_S and

n_C. The atoms $set(n_S)$ and $set(n_C)$ are added to $r(S)$ and $r(C)$ respectively. Each element of S and C is encoded by a set of rules which are added to $r(S)$ and $r(C)$, respectively. Finally, the predicate $in(.,.)$ is used to specify what belongs to S and C, respectively. Elements of C are represented by the predicates $order(*,+,+)$, $postcondition(*,+,-)$, $precondition(*,-,+)$, and $maintain(*,+,-,+)$ where the place holders '*', '+', and '-' denotes the name of a constraint, a general program, and a formula, respectively. For example, if $n_1 \prec n_2$ belongs to C then the set of atoms encoding C will contain the atom $in(order(n_0, n_1, n_2), n_C)$ where n_0 and n_C are the names assigned to the ordering constraint $n_1 \prec n_2$ and C, respectively. Similarly, if C contains (n_1, φ, n_2) then $in(maintain(n_0, n_\varphi, n_1, n_2), n_C)$ (again, n_0 and n_C are the name assigned to the truth constraint $n_1 \prec n_2$ and C, respectively) will belong to the set of atoms encoding C.

In the following example, we illustrate the encoding of a general program about the blocks world domain.

Example 7 Consider a general program, (S, C), to build a tower from blocks a, b, c that achieves the goal that a is on top of b and b is on top of c, i.e., the goal is to make $on(a, b) \wedge on(b, c)$ hold. We have $S = \{move(b, c), move(a, b)\}$, and

$$C = \left\{ \begin{array}{l} move(b,c) < move(a,b), \\ (clear(b), move(b,c)), (clear(c), move(b,c)), \\ (clear(b), move(a,b)), (clear(a), move(a,b)) \end{array} \right\}$$

With $o = move(b,c) < move(a,b)$, $f_1 = (clear(b), move(b,c))$, $f_2 = (clear(b), move(a,b))$, $f_3 = (clear(c), move(b,c))$, and $f_4 = (clear(a), move(a,b))$, the encoding of $p = (S, C)$ is as follows:

$$\begin{array}{rcl} r(p) & = & \{htn(p, n_S, n_C)\} \cup r(S) \cup r(C) \\ r(S) & = & \{set(n_S), in(move(a,b), n_S), in(move(b,c), n_S)\} \\ r(C) & = & \{set(n_C), in(o, n_C), in(f_1, n_C), in(f_2, n_C), in(f_3, n_C), in(f_4, n_C)\} \\ & & \cup\{order(o, move(b,c), move(a,b)), precondition(f_1, clear(b), move(a,b))\} \\ & & \cup\{precondition(f_2, clear(b), move(a,b))\} \\ & & \cup\{precondition(f_3, clear(c), move(b,c)), precondition(f_4, clear(a), move(a,b))\} \end{array}$$

□

We now present the LPASS rules that realize the operational semantics of general programs. For this purpose we need the rules (42)-(52) that was presented earlier. These rules are for general programs whose top level structure is not an HTN. For general programs whose top level feature is an HTN we add the following rule:

$$trans(N, T_1, T_2) \quad \leftarrow \quad htn(N, S, C), not\ nok(N, T_1, T_2). \tag{53}$$

Intuitively, the above rule states that the general program N can be unfolded between time points T_1 and T_2 (or alternatively: the trajectory from T_1 and T_2 is a trace of N) if N is an HTN construct (S, C), and it is not the case that the trajectory from T_1 and T_2 is not a trace of N. The last phrase is encoded by $nok(N, T_1, T_2)$ and is true when the trajectory from T_1 and T_2 violates one of the many constraints dictated by (S, C). The main task that now remains is to present LPASS rules that define $nok(N, T_1, T_2)$. To do that, as suggested by the definition of a trace of a program (S, C), we will need to enumerate the permutations $(i_1, \ldots, i_k)$ of $(1, \ldots, k)$ and check whether particular permutations satisfy the conditions in C. We now introduce some necessary auxiliary predicates and their intuitive meaning.

- $begin(N, I, T_3, T_1, T_2)$ – This means that I, a general program belonging to N, starts its execution at time T_3, and N starts and ends its execution at T_1 and T_2 respectively.
- $end(N, I, T_4, T_1, T_2)$ – This means that I, a general program belonging to N, ends its execution at time T_4, and N starts and ends its execution at T_1 and T_2, respectively.

- *between*(T_3, T_1, T_2) – This is an auxiliary predicate indicating that the inequalities $T_1 \leq T_3 \leq T_2$ hold.

- *not_used*(N, T, T_1, T_2) – This means that there exists no sub-program I of N whose execution covers the time moment T, i.e., $T < B$ or $T > E$ where B and E are the start and finish time of I, respectively.

- *overlap*(N, T, T_1, T_2) – This indicates that there exists at least two general programs I_1 and I_2 in N whose intervals contain T, i.e., $B_1 < T < E_1$ and $B_2 < T < E_2$ where B_i and E_i $(i = 1, 2)$ is the start- and finish-time of I_i, respectively.

We will now give the rules that define the above predicates. First, to specify that each general program I belonging to the general program (S, C), i.e., $I \in S$, must start and end its execution one and at most one time during the time (S, C) is executed, we use the following rules:

$$1\{begin(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \quad (54)$$

$$1\{end(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \quad (55)$$

The first (resp. second) rule says that I – a program belonging to S – must start (resp. end) its execution one and at most one time between T_1 and T_2. Here, we use cardinality constraints with variables [36] in expressing these constraints. Such constraints with variables are short hand for a set of instantiated rules of the form (7). For example, the first rule is a short hand for the set of rules corresponding to the following cardinality constraint:

$$1\{begin(N, I, T_1, T_1, T_2), \ldots, begin(N, I, T_2, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2).$$

We now give the rules defining *not_used*$(.,.,.,.)$ and *overlap*$(.,.,.,.)$.

$$used(N, T, T_1, T_2) \leftarrow htn(N, S, C), in(I, S), begin(N, I, B, T_1, T_2), end(N, I, E, T_1, T_2), B \leq T \leq E. \quad (56)$$

$$not\_used(N, T, T_1, T_2) \leftarrow not\ used(N, T, T_1, T_2). \quad (57)$$

$$overlap(N, T, T_1, T_2) \leftarrow htn(N, S, C), in(I_1, S), begin(N, I_1, B_1, T_1, T_2), end(N, I_1, E_1, T_1, T_2), in(I_2, S), begin(N, I_2, B_2, T_1, T_2), end(N, I_2, E_2, T_1, T_2), B_1 \leq T \leq E_1, B_2 < T < E_2. \quad (58)$$

The rule (56) states that if a general program I in N starts its execution at B and ends its execution at E then its execution spans over the interval $[B, E]$, i.e., every time moment between B and E is *used* by some general program in N. The rule (57) states that if a time moment between T_1 and T_2 is not used by some general program in N then it is *not_used*. The last rule in this group specifies the situation when two general programs belonging to N overlap.

We are now ready to define *nok*$(.,.,.)$. There are several conditions whose violation makes *nok* true. The first condition is that the time point when a program starts must occur before its finish time. Next each general program belonging to the set S of (S, C) must have a single start and finish time. The violations of these two conditions are encoded by the following rules which are added to Π.

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), in(I, S), T_3 > T_4, begin(N, I, T_3, T_1, T_2), end(N, I, T_4, T_1, T_2). \quad (59)$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), in(I, S), T_3 \leq T_4, begin(N, I, T_3, T_1, T_2), end(N, I, T_4, T_1, T_2), not\ trans(I, T_3, T_4). \quad (60)$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), T_1 \leq T \leq T_2, not\_used(N, T, T_1, T_2). \quad (61)$$

$$nok(N, T_1, T_2) \leftarrow htn(N, S, C), T_1 \leq T \leq T_2, overlap(N, T, T_1, T_2). \quad (62)$$

Together the rules (54)-(62) define when the permutation determined by the set of atoms of the form $begin(N, I, B, T_1, T_2)$ and $end(N, I, E, T_1, T_2)$ violate the initial part of condition 8 of Definition 7. the rules (54)-(55) require each general program in N to have a unique start and finish time and the rule (59) encodes the violation when the finish time is earlier than the start time. The rule (60) encodes the violation when the trace of a general program in N does not correspond to its start and finish time. The rule (61) encodes the violation when some time point on the trajectory of N is not covered by the trace of a general program in N; and the rule (62) encodes the violation when the trace of two general programs in N overlap.

The next group of rules encode the violation of the conditions 8(b) – 8(d) of Definition 7.

$$
\begin{aligned}
nok(N, T_1, T_2) &\leftarrow htn(N, S, C), in(I_1, S), begin(N, I_1, B_1, T_1, T_2), && (63)\\
&\quad\; in(I_2, S), begin(N, I_2, B_2, T_1, T_2), \\
&\quad\; in(O, C), order(O, I_1, I_2), B_1 > B_2. \\
nok(N, T_1, T_2) &\leftarrow htn(N, S, C), in(I_1, S), end(N, I_1, E_1, T_1, T_2), && (64)\\
&\quad\; in(I_2, S), begin(N, I_2, B_2, T_1, T_2), E_1 < T_3 < B_2, \\
&\quad\; in(O, C), maintain(O, F, I_1, I_2), not\ hf(F, T3). \\
nok(N, T_1, T_2) &\leftarrow htn(N, S, C), in(I, S), begin(N, I, B, T_1, T_2), && (65)\\
&\quad\; in(O, C), precondition(O, F, I), not\ hf(F, B). \\
nok(N, T_1, T_2) &\leftarrow htn(N, S, C), in(I, S), end(N, I, E, T_1, T_2), && (66)\\
&\quad\; in(O, C), postcondition(O, F, I), not\ hf(F, E).
\end{aligned}
$$

The rule (63) encodes the violation when the constraint C of the general program $N = (S, C)$ contains $I_1 \prec I_2$, but I_2 starts earlier than I_1. The rule (64) encodes the violation when C contains (I_1, F, I_2) but the formula F does not hold in some point between the end of I_1 and start of I_2. The rules (65) and (66) encode the violation when C contains the constraint (F, I) or (I, F) and F does not hold immediately before or after respectively, the execution of I.

We now formulate the correctness of our above encoding of procedural and HTN knowledge given as general programs, and relate the traces of a general program with the answer sets of its LPASS encoding. For an action theory (D, Γ) and a general program p, let Π_n^{HTN} be the LPASS program obtained from Π_n by (i) adding the rules (42)-(52) and (53)-(66), (ii) adding $r(p)$, and (iii) replacing the goal constraint with $\leftarrow not\ trans(n_P, 0, n)$. The following theorem extends Theorem 6.

Theorem 7 Let (D, Γ) be a consistent action theory and p be a general program. Then,

(i) for every answer set M of Π_n^{HTN} with $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$, $s_0(M)a_0 \ldots a_{n-1}s_n(M)$ is a trace of p; and

(ii) if $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p then there exists an answer set M of Π_n^{HTN} such that $s_j = s_j(M)$ and $occ(a_i, i) \in M$ for $j \in \{0, \ldots, n\}$ and $i \in \{0, \ldots, n-1\}$ and $trans(n_p, 0, n) \in M$.

Proof. See Appendix A.4 □

As before, to do planning using procedural and HTN constraints all we need to do is to add the goal constraint to Π_n^{HTN}, which will filter out all answer sets where the goal is not satisfied in time point n, and at the same time will use the sketch provided by the general program p.

4.4 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 [42]; and (iii) we are not aware of any empirical results indicating the utility of procedural knowledge in planning, especially in ASP. (Note that [38] concentrates on using GOLOG to do planning in domains with incomplete information, not on exploiting procedural knowledge in planning.)

We report here the result obtained from our experiment with the elevator example from [25] (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. The domain description for this example is earlier described in Example 5 and the **smodels** code can be downloaded from `http://www.cs.nmsu.edu/~tson/asp_planner`.

The initial state for this planning problem encodes a set of floors where the light is on and the current position of the elevator. For instance, $\Gamma = \{on(1), on(3), on(7), currentFloor(4)\}$. The goal formula is represented by the conjunction $\wedge_{f \text{ is a floor}} \neg on(f)$. Sometimes, the final position of the elevator is added to the goal. The planning problem is to find a sequence of actions that will serve all the floors where the light is on and thus make the *on* predicate false for all floors, and if required take the elevator to its destination floor.

Since there are a lot of plans that can achieve the desired goal, we can use procedural constraints to guide us to preferable plans. In particular, we can use the procedural knowledge encoded by the following set of simple GOLOG programs from [25], which we earlier discussed in Example 5.

$$\begin{array}{rcl} go\_floor(N) & \equiv & currentFloor(N)|up(N)|down(N). \\ serve(N) & \equiv & go\_floor(N); turnoff(N); open; close. \\ serve\_a\_floor & \equiv & \mathbf{pick}(N, on(N), serve(N)). \\ park & \equiv & \mathbf{if}\ \ currentFloor(0)\ \mathbf{then}\ \ open\ \mathbf{else}\ \ [down(0); open]. \\ control & \equiv & [\mathbf{while}\ \ \exists N.[on(N)]\ \mathbf{do}\ \ serve\_a\_floor(N)]; park \end{array}$$

We ran experiments on an HP OmniBook 6000 laptop with 130,544 Kb Ram and an Intel Pentium III 600 MHz processor, using **lparse** version 0.99.52 (Windows, build Apr 7, 2000) and **smodels** version 2.25. for planning in this example with and without the procedural control knowledge. The timings obtained are given in the following table.

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 plan length is big. 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 instances with small plan lengths, as indicated through boldface in column 6, the speed up due to the use of procedural knowledge does not make up for the overhead needed in grounding the knowledge. The output of **smodels** for each run is given in the file *result* at the above mentioned 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.

5 Conclusion

In this paper we considered three different kinds of domain dependent control knowledge (temporal, procedural and HTN-based) that are useful in planning. Our approach is declarative and relies on the language of logic programming with answer set semantics (LPASS). We showed that the addition of these three kinds of control knowledge only involves adding a few more rules to a planner written in LPASS that can plan without any control knowledge. We formally proved the correctness of our planner, both in the absence and presence of the control knowledge. Finally, we did some initial experimentation that shows the reduction in planning time when procedural domain knowledge is used and the plan length is big.

In the past temporal domain knowledge is used in planning in [1, 14]. In both cases, the planners are written in a procedural language, and there is no correctness proof of the planners. On the other hand the performance of these planners are much better[12] than our implementation using LPASS. In comparison, our focus in this paper is on the 'knowledge representation' aspects of planning with domain dependent control knowledge and demonstration of relative performance gains when such control knowledge is used. Thus we present correctness proof of our planners and stress the ease of adding the control knowledge to planner. In this regard, an interesting observation is that it is straightforward to add control knowledge from multiple sources or angles. Thus say two different general programs can be added to the planner, and any resulting plan must then satisfy the two sketches dictated by the two general programs.

As mentioned earlier our use of HTN-based constraints in planning is very different from HTN-planning and the recent HTN-based planner [33]. Unlike our approach in this paper, these planners can not be separated to two parts: one doing planning that can plan even in the absence of the knowledge encoded as HTN and the other encoding the knowledge as an HTN. In other words, these planners are not extended classical planners that allow the use of domain knowledge in the form of HTN on top of a classical planner. The timings of the planner [33] on AIPS 2000 planning domains are very good though. To convince ourselves of the usefulness of procedural constraints we used their methodology with respect to procedural domain knowledge and wrote general programs for planning with blocks world and the package delivery domain and as in [33] we wrote planners in a procedural language (the language C to be specific) for these domains and also observed similar performance. We plan to report this result in a future work. With our focus on the knowledge representation aspects we do not further discuss these experiments here.

Although we explored the use of the different kinds of domain knowledge separately, the declarativeness of our approach allows us to use the different kinds of domain knowledge for the same planning problem. For example, for a particular planning problem we may have both temporal domain knowledge and a mixture of procedural and hierarchical domain knowledge given as a general program. In that case planning will involve finding an action sequence that follows the sketch dictated by the general program and the same time obeys the temporal domain knowledge. This distinguishes our work from other related work [22, 24, 5, 32] where the domain knowledge allowed were much more restricted.

A byproduct of the way we deal with procedural knowledge is that, in a propositional environment, our approach of planning (ASP) with procedural knowledge can be viewed as an off-line interpreter for a GOLOG program. Because of the declarative nature of LPASS the correctness of this interpreter is easier to prove than the earlier interpreters which were mostly written in Prolog.

[12]This provides a challenge to the community developing LPASS systems to develop LPASS systems that can match or come close to (if not surpass) the performance of procedural systems.

Acknowledgments The first two authors would like to acknowledge the support of the NASA grant NCC2-1232. The fourth 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 grant 0070463. The work of Tran Cao Son was also supported in part by NSF grant EIA-981072.

Appendix A - Proofs

We apply the Splitting Theorem and Splitting Sequence Theorem [27] several times in our proof. For ease of reading, the basic notations and the splitting theorem are included in Appendix B. Since we assume a propositional language any rule in this paper can be considered as a collection of its ground instances. Therefore, throughout the proof, we often say a rule r whenever we refer to a ground rule r.

Appendix A.1 - Proofs of Theorem 1 and 3

It is easy to see that Theorem 1 is a special case of Theorem 3 because rules (35)-(40) are only used for formulas containing temporal operators. Thus, it suffices to prove Theorem 3.

Theorem 3 Let S be a set of goal-independent temporal formulas, $I = \langle s_0, s_1 \dots s_n \rangle$ be a sequence of states, and $I_t = \langle s_t, \dots s_n \rangle$. Let

$$\Pi = R_1 \cup R_2 \cup r(S) \cup r(I)$$

where

- R_1 consists of the set of rules (14), (15), and (28)-(33) in which the domain of T is $\{0, \dots, n\}$, the set of rules (16)-(17)), and the set of rules defining the fluents of the domain,
- R_2 is the set of rules (35)-(40) in which the domain of T is $\{0, \dots, n\}$,
- $r(I) = \cup_{t=0}^{n} \{holds(f, t) \mid l \text{ is a fluent literal and } l \in s_t\}$, and
- $r(S) = \bigcup_{\phi \in S} r(\phi)$.

Then,

(i) The program Π has a unique answer set, X.

(ii) For every temporal formula ϕ in the set S, ϕ is true in I_t, i.e., $I_t \models \phi$, if and only if $hf(n_\phi, t)$ belongs to X.

Proof. The proof is based on induction over the structural complexity of ϕ. To capture this complexity, we associate to each formula, ϕ, a non-negative number, $\sigma(\phi)$, as follows.

- $\sigma(\phi) = 0$ if ϕ is a literal.
- $\sigma(\phi) = \max_{i=1}^{k} \sigma(\phi_i) + 1$ if ϕ has the form $\mathbf{op}(\phi_1, \phi_2, \dots \phi_k)$, where **op** is a logical connective among $\neg, \wedge$ and $\vee$ (*negation*, *conj*, and *disj*), or a temporal connective **until**, **next**, **always** or **eventually**.
- $\sigma(\phi) = \sigma(\phi_1) + 1$ if $\phi = \forall X_1, \dots X_k.\phi_1$ or $\phi = \exists X_1, \dots X_k.\phi_1$.

First, we prove (i). We know that if a program is locally stratified then it has a unique answer set [4]. We will show that Π (more precisely, the set of ground rules of Π) is indeed locally stratified. To accomplish that we need to find a mapping λ from literals of Π to $\mathbf{N}$ that has the property: if $A_0 \leftarrow A_1, A_2, \ldots A_n, not\ B_1, not\ B_2, \ldots not\ B_m$ is a rule in Π, then $\lambda(A_0) \geq \lambda(A_i)$ for all $1 \leq i \leq n$ and $\lambda(A_0) > \lambda(B_j)$ for all $1 \leq j \leq m$.

We define λ as follow.

- $\lambda(nhf\_conj(\phi, t)) = 5 * \sigma(\phi) + 1$.
- $\lambda(nhf\_forall(\phi, t)) = 5 * \sigma(\phi) + 1$.
- $\lambda(nhf\_always(\phi, t)) = 5 * \sigma(\phi) + 1$.
- $\lambda(hf(\phi, t)) = 5 * \sigma(\phi) + 2$.
- $\lambda(nhf\_during(\phi, t, t')) = 5 * \sigma(\phi) + 3$.
- $\lambda(hf\_during(\phi, t, t')) = 5 * \sigma(\phi) + 4$.
- $\lambda(l) = 0$ for every other literal of Π.

Examining all the rules in Π, we can verify that λ has the necessary property.

We now prove (ii). Let X be the answer set of Π. We prove by induction over $\sigma(\phi)$.

Base: Let ϕ be a formula with $\sigma(\phi) = 0$. By the definition of σ, we know that ϕ is a literal. Then ϕ is true in s_t iff ϕ is in s_t, that is, iff $holds(\phi, t)$ belongs to X, which, because of rule (30), proves the base case since for a literal, n_ϕ is ϕ itself.

Step: Assume that for all $0 \leq j \leq k$ and formula ϕ such that $\sigma(\phi) = j$, the formula ϕ is true in s_t iff $hf(n_\phi, t)$ is in X.

Let ϕ be such a formula that $\sigma(\phi) = k + 1$.

- **Case 1:** $\phi = \neg\phi_1$. We have $\sigma(\phi_1) = \sigma(\phi) - 1 = k$. By induction, $s_t \models \phi_1$ iff $hf(n_{\phi_1}, t) \in X$. Assume $hf(n_\phi, t) \notin X$. Because of rule (29) and $negation(n_\phi, n_{\phi_1})$ being in X, we also have $hf(n_{\phi_1}, t) \in X$. It follows that $s_t \models \phi_1$, so $s_t \not\models \phi$. Now consider the case that $hf(n_\phi, t) \in X$. Formula ϕ is a negation which is supported only by the rule (29). The body of the rule is satisfied by X, so $hf(n_{\phi_1}, t) \notin X$. Hence, $s_t \not\models \phi_1$, and therefore, $s_t \models \phi$.
- **Case 2:** $\phi = \phi_1 \wedge \phi_2 \wedge \ldots \phi_i$. For all $1 \leq j \leq i$, we have $\sigma(\phi_j) \leq k$. By induction,$s_t \models \phi_i$ iff $hf(n_{\phi_i}, t) \in X$. Assume that $s_t \models \phi$, then $s_t \models \phi_i$ for all $1 \leq j \leq i$. By induction, for each j, $hf(n_{\phi_j}, t) \in X$. Thus the body of a ground rule of the form (14) with $F = n_\phi$, is not satisfiable in X. It follows that $nhf\_conj(n_\phi, t) \notin X$. Because of rule (15), we have $hf(n_\phi, t) \in X$. Now, consider the case $hf(n_\phi, t) \in X$. The only rule supporting $hf(n_\phi, t)$ is (15), so we have $nhf\_conj(n_\phi, t) \notin X$. If there exists j such that $hf(n_{\phi_j}, t) \notin X$, then the body of rule(14) is satisfied by $F = n_\phi$, $F_1 = n_{\phi_j}$, $T = t$, which causes $nhf\_conj(n_\phi, t) \in X$. Hence for all $1 \leq j \leq i$, $hf(n_{\phi_j}, t) \in X$, so $s_t \models \phi_j$. This implies $s_t \models \phi$.
- **Case 3:** $\phi = \phi_1 \vee \phi_2 \vee \ldots \phi_i$. We have $\sigma(\phi_j) \leq k$ for all $1 \leq j \leq i$, so the inductive assumption is valid for all the ϕ_j. Let $s_t \models \phi$. Then there is a ϕ_j such that $s_t \models \phi_j$. By induction, $hf(n_{\phi_j}, t) \in X$. With $F = n_\phi$, $F_1 = n_{\phi_j}$, and $T = t$ in rule (28), we have $hf(n_\phi, t) \in X$. Now assume $hf(n_\phi, t) \in X$. Because the atom is supported only by rule (28), there exists j such that $hf(n_{\phi_j}, t) \in X$. By induction, we have $s_t \models \phi_j$, so $s_t \models \phi$.

- **Case 4:** ϕ is a quantified formula. The argument for *forall*- and *exists*-quantified formula is similar to that of a conjunction and disjunction, respectively. We omit the details here for brevity.

  Before continue with the proof, observe that similar to the case of conjunctive formula, we can easily show that $hf\_during(n_\phi, t_1, t_2) \in X$ iff $hf(n_\phi, t) \in X$ (that is $s_t \models \phi$) for all $t_1 \leq t < t_2$.

- **Case 5:** $\phi = \mathbf{until}(\phi_1, \phi_2)$. We have that $\sigma(\phi_1) \leq k$ and $\sigma(\phi_2) \leq k$. Assume that $I_t \models \phi$. By Definition 2, there exists $t \leq t_2 \leq n$ such that $I_{t_2} \models \phi_2$ and for all $t \leq t_1 < t_2$, $I_{t_1} \models \phi_1$. By inductive hypothesis, $hf(n_{\phi_2}, t_2) \in X$ and for all t_1, $t \leq t_1 < t_2$, $hf(n_{\phi_1}, t_1) \in X$. It follows that $hf\_during(n_{\phi_1}, t, t_2) \in X$. Because of rule (35), we have $hf(n_\phi, t) \in X$. On the other hand, if $hf(n_\phi, t) \in X$, because the only rule supporting $hf(n_\phi, t)$ is (35), there exists $t \leq t_2 \leq n$ such that $hf\_during(n_{\phi_1}, t, t_2) \in X$ and $hf(n_{\phi_2}, t_2)$. It follows from $hf\_during(n_{\phi_1}, t, t_2) \in X$ that $hf(n_{\phi_1}, t_1) \in X$ for all $t \leq t_1 < t_2$. By inductive hypothesis, we have $I_{t_1} \models \phi_1$ for all $t \leq t_1 < t_2$ and $I_{t_2} \models \phi_2$. Thus $I_t \models \mathbf{until}(\phi_1, \phi_2)$, i.e., $I_t \models \phi$.

- **Case 6:** $\phi = \mathbf{next}(\phi_1)$. Note that $\sigma(\phi_1) \leq k$. Rule (38) is the only rule supporting $hf(n_\phi, t)$ where $\phi = \mathbf{next}(\phi_1)$. So $hf(n_\phi, t) \in X$ iff $hf(n_{\phi_1}, t+1) \in X$ iff $I_{t+1} \models \phi_1$ iff $I_t \models \mathbf{next}(\phi_1)$ iff $I_t \models \phi$.

- **Case 7:** $\phi = \mathbf{always}(\phi_1)$. We note that $\sigma(\phi_1) \leq k$. Observe that $hf(n_\phi, t)$ is supported only by rule (36). So $hf(n_\phi, t) \in X$ iff $hf\_during(n_{\phi_1}, t, n+1) \in X$. The latter happens iff $hf(n_{\phi_1}, t_1) \in X$ for all $t \leq t_1 \leq n$, that is, iff $I_{t_1} \models \phi_1$ for all $t \leq t_1 \leq n$ which is equivalent to $I_t \models \mathbf{always}(\phi_1)$, i.e., iff $I_t \models \phi$.

- **Case 8:** $\phi = \mathbf{eventually}(\phi_1)$. We know that $hf(n_\phi, t) \in X$ is supported only by rule (37). So $hf(n_\phi, t) \in X$ iff there exists $t \leq t_1 \leq n$ such that $hf(n_{\phi_1}, t_1) \in X$. Because $\sigma(\phi_1) \leq k$, by induction, $hf(n_\phi, t) \in X$ iff there exists $t \leq t_1 \leq n$ such that $I_{t_1} \models \phi_1$, that is, iff $I_t \models \mathbf{eventually}(\phi_1)$, i.e., iff $I_t \models \phi$.

The above cases prove the inductive step, and hence, the theorem. □

Appendix A.2 - Proof of Theorem 2

We now turn our attention to Theorem 2. For a planning problem $\langle D, \Gamma, \Delta \rangle$, let

$$\pi = \Pi_n(D, \Gamma, \Delta) \setminus (r(\Delta) \cup \{\leftarrow \textit{not holds}(n_\phi, n)\},$$

i.e., π is obtained from $\Pi_n(D, \Gamma, \Delta)$ by removing the rules encoding Δ and the constraint $\leftarrow \textit{not holds}(n_\Delta, n)$. The next lemma will be useful in our proofs.

Lemma 1 *For a set of causal laws K and a set of fluent literals Y, for every integer k, the program*

$$\begin{array}{lll} holds(L, k) & \leftarrow & holds(L_1, k), \ldots, holds(L_m, k) \quad (\textit{if } \mathbf{caused}(\{L_1, \ldots, L_m\}, L) \in K) \\ holds(L, k) & \leftarrow & \quad (\textit{if } L \in Y) \end{array}$$

has a unique answer set $\{holds(l, k) \mid l \in Cl_K(Y)\}$.

Proof. Let us denote the given program by P. Since P is a positive program, its unique answer set X is the least fix-point of the operator $T_P(Z) = \{l \mid \text{there exists a rule } l \leftarrow l_1, \ldots, l_n \text{ such that } l_i \in Z\}$. For a set S of literals in P, let us denote $F(S) = \{f \mid holds(f, k) \in S\}$. To prove the lemma is equivalent to prove $F(X) = Cl_K(Y)$.

We first prove that $F(X) \subseteq Cl_K(Y)$. We will show, by induction over i, that $F(T^i_P(\emptyset)) \subseteq Cl_K(Y)$ for every i. The base case is trivial since $F(T^0_P(\emptyset)) = Y \cup Y_1 \subseteq Cl_K(Y)$ where $Y_1 = \{l \mid \mathbf{causes}(\{\}, l) \in K\}$.

Assume that we have proved that $F(T_P^i(\emptyset)) \subseteq Cl_K(Y)$. We need to show that $F(T_P^{i+1}(\emptyset)) \subseteq Cl_K(Y)$. By definition of P, we have that $f \in F(T_P^{i+1}(\emptyset))$ if there exists a causal law $\mathbf{caused}(\{l_1, \ldots, l_m\}, f)$ in K such that $l_i \in F(T_P^i(\emptyset))$. This also implies that $f \in Cl_K(Y)$ because of the closeness of $Cl_K(Y)$ with respect to K. The inductive step is proved. This means that $F(X) \subseteq Cl_K(Y)$.

We now prove that $Cl_K(Y) \subseteq F(X)$. Assume that $f \in Cl_K(Y)$. By definition of $Cl_K(Y)$ we conclude that there exists a sequence of causal laws $\mathbf{caused}(\{f_1^1, \ldots, f_{n_1}^1\}, l_1)$, ..., $\mathbf{caused}(\{f_1^m, \ldots, f_{n_m}^m\}, l_m)$, where $l_m = f$ and for every t, $1 \leq t \leq m$, $\{f_1^t, \ldots, f_{n_t}^t\} \subseteq Y \cup \bigcup_{0 \leq i < t} \{f_1^i, \ldots, f_{n_i}^i\}$. It is easy to see that this implies that $f \in F(T_P^m(\emptyset))$. This means that $Cl_K(Y) \subseteq F(X)$. This, together with $F(X) \subseteq Cl_K(Y)$, proves the lemma. $\square$

We now prove some useful properties of π. We will prove that if (D, Γ) is consistent then π is consistent (i.e., π has an answer set) and that π correctly implements the transition function Φ of D. Notice that here we say that π is consistent iff π has an answer set X and for every fluent f and t, $0 \leq t \leq n$, X does not contain both $holds(f, t)$ and $holds(neg(f), t)$). First, we simplify π by using the splitting theorem [27] (Theorem 8, Appendix B). Let V be the set of literals in the language of π whose parameter list does not contain the time parameter. That is, V consists of

- atoms used to encode D: $causes(A, L, S)$, $caused(S, L)$, $in(F, F_1)$, $formula(F)$, $conj(F)$, $disj(F)$, $literal(L)$, $fluent(F)$, $action(A)$, $negation(F, F_1)$, $forall(F)$, and $exists(F)$, and
- atoms used to encode Γ: $initially(L)$.

It is easy to see that V is a splitting set of π. Furthermore, it is easy to see that the bottom program $b_V(\pi)$ consists of

- the atoms and rules encoding all the conjunctions occurring in D,
- all the ground instances of rules (16)-(19) and (22)-(27)

Obviously, $b_V(\pi)$ is a positive program, and hence, it has a unique answer set. Let us denote the unique answer set of $b_V(\pi)$ by A_0. The partial evaluation of π with respect to (V, A_0), $\pi_1 = e_V(\pi \setminus b_V(\pi), A_0)$, is the collection of the following rules[13]:

$holds(L, T{+}1)$	$\leftarrow$	$occ(A, T), hf(n_\phi, T).$	(if $\mathbf{causes}(A, L, \phi) \in D$)	(67)
$holds(L, T)$	$\leftarrow$	$hf(n_\phi, T).$	(if $\mathbf{caused}(\phi, L) \in D$)	(68)
$possible(A, T)$	$\leftarrow$	$hf(n_\phi, T).$	(if $\mathbf{executable}(a, \phi) \in D$)	(69)
$holds(L, 0)$	$\leftarrow$		(if $\mathbf{initially}(L) \in \Gamma$)	(70)
$occ(A, T)$	$\leftarrow$	$possible(A, T), not\ nocc(A, T).$	(if A is an action)	(71)
$nocc(A, T)$	$\leftarrow$	$occ(B, T).$	(for every pair of actions $A \neq B$)	(72)
$holds(F, T{+}1)$	$\leftarrow$	$holds(F, T), not\ holds(neg(F), T{+}1).$	(for every fluent F)	(73)
$holds(neg(F), T{+}1)$	$\leftarrow$	$holds(neg(F), T), not\ holds(F, T{+}1).$	(for every fluent F)	(74)
	$\leftarrow$	$holds(F, T), holds(neg(F), T).$	(for every fluent F)	(75)
$hf(F, T)$	$\leftarrow$	$hf(F_1, T).$	(if $disj(F) \in A_0$, $in(F_1, F) \in A_0$)	(76)
$nhf\_conj(F, T)$	$\leftarrow$	$not\ hf(F_1, T).$	(if $conj(F) \in A_0$, $in(F_1, F) \in A_0$)	(77)
$hf(F, T)$	$\leftarrow$	$not\ nhf\_conj(F, T).$	(if $conj(F) \in A_0$)	(78)
$hf(F, T)$	$\leftarrow$	$not\ hf(F_1, T).$	(if $negation(F, F_1) \in A_0$)	(79)
$hf(F, T)$	$\leftarrow$		(if F is a literal and $holds(F, T) \in A_0$)	(80)
$hf(F, T)$	$\leftarrow$	$hf(F_1, T).$	(if $exists(F) \in A_0, in(F_1, F) \in A_0$)	(81)
$nhf\_forall(F, T)$	$\leftarrow$	$not\ hf(F_1, T).$	(if $forall(F) \in A_0$, $in(F_1, F) \in A_0$)	(82)
$hf(F, T)$	$\leftarrow$	$not\ nhf\_forall(F, T).$	(if $forall(F) \in A_0$)	(83)

[13] Again, a rule with variables stands for a collection of its ground instances. Further, ϕ stands for a conjunction of literals occurring in D.

Let π_2 be the program consisting of the following rules:

$$\begin{array}{rcll}
holds(L,T{+}1) & \leftarrow & occ(A,T), holds(L_1,T),\dots,holds(L_k,T). & (84) \\
 & & \qquad (\text{if } \mathbf{causes}(A,L,\{L_1,\dots,L_k\}) \in D) & \\
holds(L,T) & \leftarrow & holds(L_1,T),\dots,holds(L_m,T). \quad (\text{if } \mathbf{caused}(\{L_1,\dots,L_m\},L) \in D) & (85) \\
possible(A,T) & \leftarrow & holds(L_1,T),\dots,holds(L_t,T). \quad (\text{if } \mathbf{executable}(a,\{L_1,\dots,L_t\}) \in D) & (86) \\
holds(L,0) & \leftarrow & \qquad (\text{if } \mathbf{initially}(L) \in \Gamma) & (87) \\
occ(A,T) & \leftarrow & possible(A,T), not\ nocc(A,T). \quad (\text{if } A \text{ is an action}) & (88) \\
nocc(A,T) & \leftarrow & occ(B,T). \quad (\text{for every pair of actions } A \neq B) & (89) \\
holds(F,T{+}1) & \leftarrow & holds(F,T), not\ holds(neg(F),T{+}1). \quad (\text{for every fluent } F) & (90) \\
holds(neg(F),T{+}1) & \leftarrow & holds(neg(F),T), not\ holds(F,T{+}1). \quad (\text{for every fluent } F) & (91) \\
 & \leftarrow & holds(F,T), holds(neg(F),T). \quad (\text{for every fluent } F)^{14} & (92)
\end{array}$$

Because of Theorem 3, it is easy to see that the following lemma holds.

Lemma 2 *For answer set X of π_1, $Y = X \cap lit(\pi_2)$ is an answer set of π_2.*

For every answer set Y of π_2, $X = Y \cup \{hf(n_\phi,t) \mid formula(n_\phi) \in A_0,\ s_t(Y) \models \phi\}$ is an answer set of π_1 where $s_t(Y) = \{l \mid l \text{ is a fluent literal and } holds(l,t) \in Y\}$. □

It follows from the splitting theorem and from Lemma 2 that to prove the consistency and correctness of π it is enough to prove the consistency of π_2, the program consisting of rules (84)-(91) and that π_2 correctly implements the transition function Φ of D. We prove this in the next lemmas.

Lemma 3 *Let X be an answer set of π_2. Then, for every t, $0 \le t \le n$,*

1. *$s_t(X)$ is a state of D,*
2. *if X contains $occ(a,t)$ then a is executable in $s_t(X)$ and $s_{t+1}(X) \in \Phi(a, s_t(X))$[15], and*
3. *if $occ(a,t) \notin X$ for every action a, then $s_{t+1}(X) = s_t(X)$.*

Proof. It is easy to see that the sequence $\langle U_t \rangle_{t=0}^n$, where

$$\begin{array}{rcl}
U_t & = & \{holds(L,T) \mid L \text{ is a literal and } T \le t\} \cup \{occ(A,T) \mid A \text{ is an action and } T \le t\} \cup \\
 & & \{nocc(A,T) \mid A \text{ is an action and } T \le t\} \cup \{possible(A,T) \mid A \text{ is an action and } T \le t\},
\end{array}$$

is a splitting sequence of π_2. Since X is an answer set of π_2, by the splitting sequence theorem (Theorem 9, Appendix B), there exists a sequence of sets of literals $\langle X_t \rangle_{t=0}^n$ such that $X_t \subseteq U_t \setminus U_{t-1}$, and

- $X = \bigcup_{i=0}^n X_i$,
- X_0 is an answer set of $b_{U_0}(\pi_2)$ and
- for every $t > 0$, X_t is an answer set of $e_{U_t}(b_{U_t}(\pi_2) \setminus b_{U_{t-1}}(\pi_2), \bigcup_{i \le t-1} X_i)$.

We will prove the lemma by inductively proving that for every t, $0 \le t \le n$, X_t satisfies the following conditions:

[14] It is easy to see that every answer set of π_2 is an answer set of the program consisting of rules (84)-(91) that satisfies the constraint (92). For this reason, rule (92) will be omitted subsequently when we use the splitting theorem.

[15] Recall that for every set Y, $s_t(Y)$ is the set $\{f \mid holds(f,t) \in Y\}$

(i) every X_t is complete and consistent with respect to $\mathbf{F}$ in the sense that for each fluent f, X_t contains either $holds(f,t)$ or $holds(neg(f),t)$ but not both,

(ii) every X_t contains at most one atom of the form $occ(a,t)$,

(iii) $s_t(X_t)$ is a state of D, and

(iv) if $occ(a,t-1) \in X_{t-1}$ then a is executable in $s_{t-1}(X_{t-1})$ and $s_t(X_t) \in \Phi(a, s_{t-1}(X_{t-1}))$; if no atom of the form $occ(a,t-1)$ belongs to X_{t-1} then $s_{t-1}(X_{t-1}) = s_t(X_t)$.

Base case: $t = 0$. Trivially, X_0 satisfies (iv). So, we only need to show that X_0 satisfies (i)-(iii). Let $P_0 = b_{U_0}(\pi_2)$. We have that P_0 consists of only rules of the form (85)-(89) in which $T = 0$. Let $Z_0 = \{holds(f,0) \mid f \text{ is a fluent}\} \cup \{holds(neg(f),0) \mid f \text{ is a fluent}\}$. We can easily checked that Z_0 is a splitting set of P_0. Thus, by the splitting theorem, $X_0 = M_0 \cup N_0$ where M_0 is an answer set of $b_{Z_0}(P_0)$ and N_0 is an answer set of $e_{Z_0,M_0} = e_{Z_0}(P_0 \setminus b_{Z_0}(P_0), M_0)$. Because M_0 contains only literals of the form $holds(f,0)$ and N_0 contains only literals of the form $occ(a,0)$, $nocc(a,0)$, and $possible(a,0)$, we have that $s_0(X_0) = s_0(M_0)$ and $occ(a,0) \in M_0$ iff $occ(a,0) \in N_0$. Hence, to prove that X_0 satisfies (i)-(iv), we show that M_0 satisfies (i) and (iii) and N_0 satisfies (ii).

We have that the bottom program $b_{Z_0}(P_0)$ consists of rules of the form (87) and (85). It follows from Lemma 1 that M_0 is the unique answer set of $b_{Z_0}(P_0)$ and $M_0 = \{holds(f,0) \mid f \in s_0\}$ where s_0 is the initial state of (D,Γ). Because of the completeness of Γ and the consistency of (D,Γ), we can conclude that M_0 is complete and consistent. Thus, M_0 satisfies (i). Furthermore, because $s_0(M_0) = s_0$, we conclude that M_0 satisfies (iii).

The partial evaluation of P_0 with respect to (Z_0, M_0), e_{Z_0,M_0}, consists of

$$
e_{Z_0,M_0} = \left\{
\begin{array}{lcll}
possible(A,0) & \leftarrow & & (a1) \\
 & & (\text{if } \mathbf{executable}(A, \{L_1,\ldots,L_m\}) \in D \text{ and } holds(L_i,0) \in M_0) & \\
occ(A,0) & \leftarrow & possible(A,0), not\ nocc(A,0). & (a2) \\
nocc(A,0) & \leftarrow & occ(B,0). & (a3) \\
 & & (\text{for every pair of actions } A \neq B) & \\
 & \leftarrow & holds(F,0), holds(neg(F),0) & (a4) \\
 & & (\text{for every fluent } F) &
\end{array}
\right.
$$

Let R be the set of atoms occurring in the rule (a1) of e_{Z_0,M_0}. There are two cases:

- **Case 1:** $R = \emptyset$. Obviously, the empty set is the unique answer set of e_{Z_0,M_0}. Thus, N_0 does not contain any atom of the form $occ(a,0)$.
- **Case 2:** $R \neq \emptyset$. By applying the splitting theorem one more time with the splitting set R we can conclude that N_0 is an answer set of e_{Z_0,M_0} if and only if there exists some action a, $possible(a,0) \in R$, and

$$N_0 = R \cup \{occ(a,0)\} \cup \{nocc(b,0) \mid b \text{ is an action in } D, b \neq a\}.$$

  Thus, N_0 contains only one atom of the form $occ(a,0)$.

The above two cases show that N_0 contains at most one atom of the form $occ(a,0)$. This concludes the proof of the base case.

Step: Assume that X_t, $t < k$, satisfies (i)-(iv). We will show that X_k also satisfies (i)-(iv). Let $M_{k-1} = \bigcup_{t=0}^{k-1} X_t$. The splitting sequence theorem implies that X_k is an answer set of P_k that consists of the following rules:

$$holds(L,k) \leftarrow \qquad (93)$$
$$(\text{if } occ(A,k-1) \in M_{k-1},\ \mathbf{causes}(A,L,\{L_1,\ldots,L_k\}) \in D,\ holds(L_i,k-1) \in M_{k-1})$$

$$
\begin{array}{rcll}
holds(L,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_m,k). & (\text{if } \mathbf{caused}(\{L_1,\ldots,L_m\},L)\in D) \quad (94)\\
possible(A,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_t,k). & (\text{if } \mathbf{executable}(a,\{L_1,\ldots,L_t\})\in D) \quad (95)\\
occ(A,k) & \leftarrow & possible(A,k), not\ nocc(A,k). & (\text{if } A \text{ is an action}) \quad (96)\\
nocc(A,k) & \leftarrow & occ(B,k). & (\text{for every pair of actions } A\neq B) \quad (97)\\
holds(F,k) & \leftarrow & not\ holds(neg(F),k). & (\text{ if } holds(F,k-1)\in M_{k-1}) \quad (98)\\
holds(neg(F),k) & \leftarrow & not\ holds(F,k). & (\text{ if } holds(neg(F),k-1)\in M_{k-1}) \quad (99)\\
 & \leftarrow & holds(F,k), holds(neg(F),k). & \quad (100)
\end{array}
$$

There are two cases:

- **Case 1:** M_{k-1} does not contain an atom of the form $occ(a,k-1)$. Then, it is easy to check that $s_k(X_k)=s_{k-1}(X_{k-1})$. Because X_{k-1} satisfies (i)-(iv), X_k also satisfies (i)-(iv).
- **Case 2:** There exists an action a such that $occ(a,k-1)\in M_{k-1}$. Let $s'=\{l \mid holds(l,k)\in X_k\}$. From the constraint (100), we have that for every fluent f, X_k cannot contain both $holds(f,k)$ and $holds(neg(f),t)$. This means that X_k is consistent. We now show that X_k is also complete. Assume the contrary, i.e., there exists a fluent f such that neither $holds(f,k)$ nor $holds(neg(f),f)$ belongs to X_k. Because of the completeness of $s_{k-1}(X_{k-1})$ (Item (i), inductive hypothesis), either $holds(f,k-1)\in s_{k-1}(X_{k-1})$ or $holds(f,k-1)\notin s_{k-1}(X_{k-1})$. If the first case happens, rule (98) belongs to P_k, and hence, X_k must contain $holds(f,k)$, which contradicts our assumption that $holds(f,k)\notin X_k$. Similarly, if the second case happens, because of rule (99), we can conclude that $holds(neg(f),k)\in X_k$ which is also a contradiction. Thus, our assumption on the incompleteness of X_k is incorrect. In other words, we have proved that X_k is indeed complete and consistent, i.e., (i) is proved for X_k.

  Let $Y_k=\{holds(l,k)\mid l \text{ is a fluent literal and } holds(l,k)\in X_k\}$ and $Z_k=\{holds(l,k)\mid l \text{ is a fluent literal}\}$. Z_k is a splitting set of P_k. Let $\pi_k=b_{Z_k}(P_k)$. From the splitting theorem, we know that Y_k must be an answer set of the program $(\pi_k)^{Y_k}$ that consists of the following rules:

$$
\begin{array}{rcll}
holds(L,k) & \leftarrow & (\text{if } occ(A,k-1)\in M_{k-1}, & \mathbf{causes}(A,L,\{L_1,\ldots,L_k\})\in D, \quad (b1)\\
 & & & holds(L_i,k-1)\in M_{k-1})\\
holds(L,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_m,k). & (\text{if } \mathbf{caused}(\{L_1,\ldots,L_m\},L)\in D) \quad (b2)\\
holds(F,k) & \leftarrow & (\text{ if } holds(F,k-1)\in M_1 & \text{and } holds(neg(F),k)\notin Y_k) \quad (b3)\\
holds(neg(F),k) & \leftarrow & (\text{ if } holds(neg(F),k-1)\in M_1 & \text{and } holds(F,k)\notin Y_k) \quad (b4)
\end{array}
$$

  Let Q_1 and Q_2 be the set of atoms occurring in the rule (b1) and (b3)-(b4), respectively. Let $C_1=\{l\mid holds(l,k)\in Q_1\}$ and $C_2=\{l\mid holds(l,k)\in Q_2\}$. By definition of Y_k, Q_1, and Q_2, we can conclude that $C_1=E(a,s_{k-1}(X_{k-1}))$ and $C_2=s'\cap s_{k-1}(X_{k-1})$. Furthermore, Lemma 1 implies that $(\pi_k)^{Y_k}$ has a unique answer set $\{holds(f,k)\mid f\in Cl_{D_C}(C_1\cup C_2)\}$ which is Y_k (because Y_k is an answer set of π_k). Hence, $s'=Cl_{D_C}(E(a,s_{k-1}(X_{k-1}))\cup(s'\cap s_{k-1}(X_{k-1})))$. This implies that $s'\in\Phi(a,s_{k-1}(X_{k-1}))$. In other words, we have proved that X_k satisfies (iii)-(iv).

The above two cases show that X_k satisfies (iii) and (iv). It remains to be shown that X_k contains at most one atom of the form $occ(a,k)$. Again, by the splitting theorem, we can conclude that $N_k=X_k\setminus Y_k$ must be an answer set of the following program

$$
e_{Y_k}=\left\{
\begin{array}{lcll}
possible(A,k) & \leftarrow & (\text{if } \mathbf{executable}(A,\{L_1,\ldots,L_m\})\in D \text{ and } holds(L_i,k)\in Y_k)\\
occ(A,k) & \leftarrow & possible(A,k), not\ nocc(A,k). & (\text{if } A \text{ is an action})\\
nocc(A,k) & \leftarrow & occ(B,k). & (\text{for every pair of actions } A\neq B)
\end{array}\right.
$$

Let R_k be the set of atoms occurring in the first rule of e_{Y_k}. Similar to the proof of the base case, we can show that for every answer set N_k of e_{Y_k}, either N_k does not contain an atom of the form $occ(a,k)$ of there exists one and only one action a such that $possible(a,k)\in R_k$ and $N_K=R_k\cup\{occ(a,k)\}\cup\{nocc(b,a)\mid b$ is an action, $b\neq a\}$. In either case, we have that $X_k=Y_k\cup N_k$ satisfies the conditions (ii). The inductive step is proved.

The conclusion of the lemma follows immediately from the fact that $s_t(X)=s_t(X_t)$ for every t and $occ(a,t)\in X$ iff $occ(a,t)\in X_t$ and X_t satisfies the property (i)-(iv). The lemma is proved. □

Lemma 4 *For every trajectory $s_0a_0\ldots a_{n-1}s_n$ in D and a consistent action theory (D,Γ), π_2 has an answer set X such that for every t, $0 \le t \le n$,*

1. *$s_t(X) = s_t$ and*
2. *$occ(a_t,t) \in X$.*

Proof. We prove the theorem by constructing an answer set X of π_2 that satisfies the Items 1 and 2. Again, we apply the splitting sequence theorem with the splitting sequence $\langle U_t\rangle_{t=0}^{n}$, where

$$\begin{array}{rcl} U_t & = & \{holds(L,T) \mid L \text{ is a literal and } T \le t\} \cup \{occ(A,T) \mid A \text{ is an action and } T \le t\} \cup \\ & & \{nocc(A,T) \mid A \text{ is an action and } T \le t\} \cup \{possible(A,T) \mid A \text{ is an action and } T \le t\}. \end{array}$$

We will show that the following sequence of sets of literals $\langle X_t\rangle_{t=0}^{n}$,

$$X_t = \{holds(f,t) \mid f \in s_t\} \cup \{occ(a_t,t)\} \cup \{nooc(b,t) \mid b \text{ is an action in } D,\ b \neq a\} \cup R_t,$$

where $R_t = \{possible(a,t) \mid a \text{ is executable in } s_t\}$ is a solution to π_2 with respect to $\langle U_t\rangle_{t=0}^{n}$. This amounts to prove that

- X_0 is an answer set of $b_{U_0}(\pi_2)$ and
- for every $t > 0$, X_t is an answer set of $e_{U_t}(b_{U_t}(\pi_2) \setminus b_{U_{t-1}}(\pi_2), \bigcup_{i \le t-1} X_i)$.

We first prove that X_0 is an answer set of $P_0 = b_{U_0}(\pi_2)$. By the construction of P_0 and X_0, we have that $(P_0)^{X_0}$ consists of the following rules:

$$(P_0)^{X_0} = \left\{ \begin{array}{lcll} holds(F,0) & \leftarrow & (\text{if } \mathbf{initially}(f) \in \Gamma) & (a1) \\ holds(L,0) & \leftarrow & holds(L_1,0),\ldots,holds(L_m,0). & (a2) \\ & & (\text{if } \mathbf{caused}(\{L_1,\ldots,L_m\},L) \in D) & \\ possible(A,0) & \leftarrow & holds(L_1,0),\ldots,holds(L_m,0). & (a3) \\ & & (\text{if } \mathbf{executable}(A,\{L_1,\ldots,L_m\}) \in D) & \\ occ(a_0,0) & \leftarrow & possible(a_0,0). & (a4) \\ nocc(B,0) & \leftarrow & occ(A,0). & (a5) \\ & & (\text{for every pair of actions } B \neq A) & \\ & \leftarrow & holds(F,0), holds(neg(F),0) & (a6) \\ & & (\text{for every fluent } F) & \end{array} \right.$$

We will show that X_0 is a minimal set of literals closed under the rules (a1)-(a6) and therefore is an answer set of P_0. Since $holds(f,0) \in X_0$ iff $f \in s_0$ (Definition of X_0) and $f \in s_0$ iff $\mathbf{initially}(f) \in \Gamma$ (Definition of s_0), we conclude that X_0 is closed under the rule (a1). Because of s_0 is closed under the static causal laws in D, we conclude that X_0 is closed under the rule (a2). The definition of R_0 guarantees that X_0 is closed under the rule (a3). Since $s_0a_0\ldots a_{n-1}s_n$ is a trajectory of D, a_0 is executable in S_0. This implies that $possible(a_0,0) \in R_0$. This, together with the fact that $occ(a_0,0) \in X_0$, implies that X_0 is closed under the rule (a4). The construction of X_0 also implies that X_0 is closed under the rule (a5). Finally, because of the consistency of Γ, we have that X_0 does not contain $holds(f,0)$ and $holds(neg(f),0)$ for any fluent f. Thus, X_0 is closed under the rules (a1)-(a6).

To complete the proof, we need to show that X_0 is minimal. Consider an arbitrary set of atoms X' that is closed under the rules (a1)-(a6). This implies the following:

- $holds(f,0) \in X'$ for every $f \in s_0$ (because of the rule (a1)).
- $R_0 \subset X'$ (because of the rule (a3) and the definition of R_0).
- $occ(a_0,0) \in X'$ (because of the rule (a4)).
- $\{nocc(b,0) \mid b \text{ is an action, } b \neq a\} \subseteq X'$ (because $occ(a_0,0) \in X'$ and the rule (a5)).

The above items imply that $X_0 \subseteq X'$. In other words, we show that X_0 is a minimal set of literals that is closed under the rules (a1)-(a6). This implies that X_0 is an answer set of $(P_0)^{X_0}$, which implies that X_0 is an answer set of P_0.

To complete the proof of the lemma, we will prove by induction over t, $t > 0$, that X_t is an answer set of $P_t = e_{U_t}(b_{U_{t-1}}(\pi_2) \setminus b_{U_{t-1}}(\pi_2), \bigcup_{i \le t-1} X_i)$. Since the proof of the base case ($t = 1$) and the inductive step is similar, we skip the base case and present only the proof for the inductive step. Now, assuming that X_t, $t < k$, is an answer set of P_t. We show that X_k is an answer set of P_k. Let $M_{k-1} = \bigcup_{i \le k-1} X_i$. The program P_k consists of the following rules:

$$
\begin{array}{rcll}
holds(L,k) & \leftarrow & (\text{if } occ(A,k-1) \in M_{k-1},\ \mathbf{causes}(A,L,\{L_1,\ldots,L_k\}) \in D, & (101)\\
 & & \quad holds(L_i,k-1) \in M_{k-1}) & \\
holds(L,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_m,k). \qquad (\text{if } \mathbf{caused}(\{L_1,\ldots,L_m\},L) \in D) & (102)\\
possible(A,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_t,k). \qquad (\text{if } \mathbf{executable}(a,\{L_1,\ldots,L_t\}) \in D) & (103)\\
occ(A,k) & \leftarrow & possible(A,k), not\ nocc(A,k). \qquad (\text{if } A \text{ is an action}) & (104)\\
nocc(A,k) & \leftarrow & occ(B,k). \qquad (\text{for every pair of actions } A \neq B) & (105)\\
holds(F,k) & \leftarrow & not\ holds(neg(F),k). \qquad (\text{ if } holds(F,k-1) \in M_{k-1}) & (106)\\
holds(neg(F),k) & \leftarrow & not\ holds(F,k). \qquad (\text{ if } holds(neg(F),k-1) \in M_{k-1}) & (107)\\
 & \leftarrow & holds(F,k), holds(neg(F),k). & (108)
\end{array}
$$

It is easy to see that P_k can be split by the set of literal $Z_k = \{holds(f,k) \mid f \text{ is a fluent literal}\}$ and the bottom program $\pi_k = b_{Z_k}(P_k)$ consists of the rules (101)-(102) and (106)-(107). We will prove first that $Y_k = \{holds(l,k) \mid holds(l,k) \in X_k\}$ is an answer set of the program $(\pi_k)^{Y_k}$ that consists of the following rules:

$$
\begin{array}{lcll}
holds(L,k) & \leftarrow & (\text{if } occ(A,k-1) \in M_{k-1}, \quad \mathbf{causes}(A,L,\{L_1,\ldots,L_k\}) \in D, & (b1)\\
 & & \qquad\qquad holds(L_i,k-1) \in M_{k-1}) & \\
holds(L,k) & \leftarrow & holds(L_1,k),\ldots,holds(L_m,k). \quad (\text{if } \mathbf{caused}(\{L_1,\ldots,L_m\},L) \in D) & (b2)\\
holds(F,k) & \leftarrow & (\text{ if } holds(F,k-1) \in M_1 \quad \text{and } holds(neg(F),k) \notin Y_k) & (b3)\\
holds(neg(F),k) & \leftarrow & (\text{ if } holds(neg(F),k-1) \in M_1 \quad \text{and } holds(F,k) \notin Y_k) & (b4)
\end{array}
$$

Let Q_1 and Q_2 be the set of atoms occurring in the rule (b1) and (b3)-(b4), respectively. Let $C_1 = \{l \mid holds(l,k) \in Q_1\}$ and $C_2 = \{l \mid holds(l,k) \in Q_2\}$. By definition of Y_k, Q_1, and Q_2, we can conclude that $C_1 = E(a, s_{k-1}(X_{k-1}))$ and $C_2 = s_k \cap s_{k-1}(X_{k-1})$. Furthermore, Lemma 1 implies that $(\pi_k)^{Y_k}$ has a unique answer set $\{holds(f,k) \mid f \in Cl_{D_C}(C_1 \cup C_2)\} = \{holds(f,k) \mid f \in s_k\}$ which equals Y_k because of the construction of X_k and Y_k. Thus, Y_k is an answer set of π_k. It follows from the splitting theorem that to complete the proof of the inductive step, we need to show that $N_k = X_k \setminus Y_k$ is an answer set of the partial evaluation of P_k with respect to (Z_k, Y_k), $e_{Z_k,Y_k} = e_{Z_k}(P_k \setminus b_{Z_k}(P_k), X_k)$, which is the following program

$$
e_{Z_k,Y_k} = \left\{
\begin{array}{lcll}
possible(A,k) & \leftarrow & (\text{if } \mathbf{executable}(A,\{L_1,\ldots,L_m\}) \in D \text{ and } holds(L_i,k) \in Y_k) & \\
occ(A,k) & \leftarrow & possible(A,k), not\ nocc(A,k). & (\text{if } A \text{ is an action})\\
nocc(A,k) & \leftarrow & occ(B,k). & (\text{for every pair of actions } A \neq B)
\end{array}
\right.
$$

It is easy to see that the reduct of e_{Z_k,Y_k} with respect to N_k, $(e_{Z_k,Y_k})^{N_k}$, consists of the following rules

$$
(e_{Z_k,Y_k})^{N_k} = \left\{
\begin{array}{lcll}
possible(A,k) & \leftarrow & (\text{if } \mathbf{executable}(A,\{L_1,\ldots,L_m\}) \in D \text{ and } holds(L_i,k) \in Y_k) & \\
occ(a_k,k) & \leftarrow & possible(a_k,k). & \\
nocc(A,k) & \leftarrow & occ(B,k). & (\text{for every pair of actions } A \neq B)
\end{array}
\right.
$$

Let R_k be the set of atoms occurring in the first rule of $(e_{Z_k,Y_k})^{N_k}$. Because $s_0 a_0 \ldots a_n s_n$ is a trajectory in D, a_k is executable in s_k. Thus, $possible(a_k, k)$ belongs to R_k. It is easy to see that N_k is the unique answer set of $(e_{Z_k,Y_k})^{N_k}$. In other words, N_k is an answer set of e_{Z_k,Y_k}. The inductive step is proved.

The property of X_t implies that the sequence $\langle X_t \rangle_{t=0}^n$ is a solution to π_2 with respect to the sequence $\langle U_t \rangle_{t=0}^n$. By the splitting sequence theorem, $X = \bigcup_{t=0}^n X_t$ is an answer set of π_2. Because of the construction of X_t, we have that $s_t(X) = s_t(X_t) = s_t$ for every t and $occ(a_t, t) \in X$ for every t, $0 \le t \le n$. The lemma is proved. $\square$

The above lemmas lead to the following corollaries.

Corollary 5.1 *Let X be an answer set of π. Then, for every t, $0 \leq t \leq n$,*

(i) $s_t(X)$ is a state of D,

(ii) if X contains $occ(a,t)$ then a is executable in $s_t(X)$ and $s_{t+1}(X) \in \Phi(a, s_t(X))$, and

(iii) if $occ(a,t) \notin X$ for every action a, then $s_{t+1}(X) = s_t(X)$.

Proof. It follows from the Lemma 2 that $Y = X \cap lit(\pi_2)$ is an answer set of π_2. Because $s_t(X) = s_t(Y)$ and Lemma 3, we conclude that X satisfies the (i)-(iii). $\square$

Corollary 5.2 *For every trajectory $s_0a_0 \ldots a_{n-1}s_n$ in D and a consistent action theory (D, Γ), π has an answer set X such that for every t, $0 \leq t \leq n$,*

(i) $s_t(X) = s_t$ and

(ii) $occ(a_t, t) \in X$.

Proof. From Lemma 4, there exists an answer set Y of π_2 such that $s_t(Y) = s_t$ and $occ(a_t,t) \in Y$. It follows from the Lemma 2 that $X = Y \cup \{hf(n_\phi, t) \mid formula(n_\phi) \in A_0, s_t(Y) \models \phi\}$ is an answer set of π. Because $s_t(X) = s_t(Y)$, we conclude that X satisfies (i)-(ii). $\square$

The next observation is also useful.

Observation 5.1 *For every answer set X of π, if there exists an t such that X does not contain an atom of the form $occ(a,t)$, then X does not contain an atom of the form $occ(a,t')$ for $t \leq t'$.*

Using the result of Theorem 3 and the above corollaries we can prove Theorem 2.

Theorem 2 For a planning problem $\langle D, \Gamma, \Delta \rangle$,

(i) if $s_0a_0 \ldots a_{n-1}s_n$ is a trajectory achieving Δ, then there exists an answer set M of Π_n such that

1. $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$ and
2. $s_i = s_i(M)$ for $i \in \{0, \ldots, n\}$.

and

(ii) if M is an answer set of Π_n, then there exists an integer $0 \leq k \leq n$ such that $s_0(M)a_0 \ldots a_{k-1}s_k(M)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$. Moreover, if $k < n$ then no action is executable in the state $s_k(M)$.

Proof. We have that $\Pi_n = \pi \cup r(\Delta) \cup \{\leftarrow not\ hf(n_\Delta, n)\}$. It is easy to see that Π_n can be split by $U = lit(\pi)$, the set of literals in the language of π, and that π is the bottom, $b_U(\Pi_n)$. Thus, M is an answer set of Π_n iff $M = X \cup Y$ where X is an answer set of π and Y is an answer set of $e_U(\Pi_n \setminus b_U(\Pi_n), X)$.

(i). Since $s_0a_0 \ldots a_{n-1}s_n$ is a trajectory achieving Δ, which is also a trajectory in D, the existence of X that satisfies the condition (i) of the Lemma follows form Corollary 5.2. Theorem 3 implies that $hf(n_\Delta, n)$ belongs to X because $s_n \models \Delta$. Thus, $e_U(\Pi_n \setminus b_U(\Pi_n), X)$ only contains rules belonging to

$r(\Delta)$ which is clearly a consistent program, i.e., it has an answer set Y. This implies the existence of M satisfying (i).

(ii). Let M be an answer set of Π_n. Then, $X = M \setminus (r(\Delta) \setminus lit(\pi))$ is an answer set of π. It follows from Observation 5.1 that there exists an integer $k \leq n$ such that for each i, $0 \leq i < k$, there exists an action a_i such that $occ(a_i, i) \in M$ and for $t \geq k$, $occ(a, t) \notin M$ for every action a. By Corollary 5.1, we know that a_i is executable in $s_i(M)$ and $s_{i+1}(M) \in \Phi(a_i, s_i(M))$. This means that $s_0(M)a_0 \ldots a_{k-1}s_k(M)$ is a trajectory (D, Γ) and $s_k(M) = s_n(M)$. Moreover, $hf(n_\Delta, n)$ must be in M, otherwise $e_U(\Pi_n \setminus b_U(\Pi_n), X)$ contains the constraint $\leftarrow$ *not* $hf(n_\Delta, n)$ which causes it to be inconsistent, which contradicts the fact that M is an answer set of Π_n. This, together with Theorem 3, implies that Δ holds in $s_n(M) = s_k(M)$. Thus, $s_0(M)a_0 \ldots a_{k-1}s_k(M)$ is a trajectory achieving Δ. Furthermore, it follows from Corollary 5.1 and the rules (88) and (89) that if $k < n$ then M does not contain literals of the form $possible(a, k)$. This implies that no action is executable in $s_k(M)$. □

Appendix A.3 - Proof of Theorem 6

We first prove some lemmas that are needed for proving Theorem 6.

Lemma 5 *For a consistent action theory (D, Γ), a program p, and an answer set M of Π_n^T with $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$, $s_0(M)a_0s_1(M) \ldots a_{n-1}s_n(M)$ is a trace of p.*

Proof. It is easy to see to see that the union of the set of literals of π and the set of rules and atoms encoding p, i.e., $U = lit(\pi) \cup r(p)$, is a splitting set of Π_n^T. Further, $b_U(\Pi_n^T) = \pi \cup r(p)$. Thus, by the splitting theorem, M is an answer set of Π_n^T iff $M = X \cup Y$ where X is an answer set of $\pi \cup r(P)$, and Y is an answer set of $e_U(\Pi_n^T \setminus \pi, X)$. Because of the constraint $\leftarrow$ *not* $trans(n_P, 0, n)$, we know that if M is an answer set of Π_n^T then every answer set Y of $e_U(\Pi_n^T \setminus \pi, X)$ must contain $trans(n_P, 0, n)$. Furthermore, we have that $s_t(X) = s_t(M)$ for every t. Hence, in what follows we will use $s_t(X)$ and $s_t(M)$ interchangeably. We prove the conclusion of the lemma by proving a stronger conclusion[16]:

(*) for every program q occurring in p and two time points t_1, t_2 such that $q \neq$ **null** and $trans(n_q, t_1, t_2) \in M$, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q (the states $s_i(M)$ and actions a_i are given in the Lemma's statement).

Denote $\pi_1 = e_U(\Pi_n^T \setminus \pi, X)$. We have that π_1 consists of the following rules:

$$trans(A, T, T+1) \leftarrow \text{(if } action(A) \in X,\ occ(A, T) \in X) \tag{109}$$

$$trans(F, T_1, T_1) \leftarrow \text{(if } formula(F) \in X,\ hf(F, T_1) \in X) \tag{110}$$

$$trans(P, T_1, T_2) \leftarrow T1 \leq T' \leq T_2, trans(P_1, T_1, T'), trans(P_2, T', T_2). \quad \text{(if } proc(P, P_1, P_2) \in X)) \tag{111}$$

$$trans(N, T_1, T_2) \leftarrow trans(P_1, T_1, T_2). \quad \text{(if } choiceAction(N) \in X,\ in(P_1, N) \in X) \tag{112}$$

$$trans(I, T_1, T_2) \leftarrow trans(P_1, T_1, T_2). \quad \text{(if } if(I, F, P_1, P_2) \in X,\ hf(F, T_1) \in X) \tag{113}$$

$$trans(I, T_1, T_2) \leftarrow trans(P_2, T_1, T_2). \quad \text{(if } if(I, F, P_1, P_2) \in X,\ hf(F, T_1) \notin X) \tag{114}$$

$$trans(W, T_1, T_2) \leftarrow T_1 < T' \leq T_2, trans(P, T_1, T'), trans(W, T', T_2). \quad \text{(if } while(W, F, P) \in X,\ hf(F, T_1) \in X) \tag{115}$$

$$trans(W, T, T) \leftarrow \text{(if } while(W, F, P) \in X, hf(F, T) \notin X) \tag{116}$$

[16] Recall that for simplicity, in encoding programs or formulas we use l or a as the name associated to l or a, respectively.

$$trans(S, T_1, T_2) \leftarrow trans(P, T_1, T_2). \qquad (117)$$
$$\text{(if } choiceArgs(S, F, P) \in X, hf(F, T_1) \in X)$$
$$trans(\mathbf{null}, T, T) \leftarrow \qquad (118)$$

Clearly, π_1 is a positive program. Thus, the unique answer set of π_1 is the fix-point of the T_{π_1} operator, defined by $T_{\pi_1}(X) = \{A \mid \text{there exists a rule } A \leftarrow A_1, \ldots, A_n \text{ in } \pi_1 \text{ such that } A_i \in X\}$. Let $Y_k = T^k_{\pi_1}(\emptyset)$. By definition $Y = \lim_{n \to \infty} Y_n$.

For every atom $A \in Y$, let $\rho(A)$ denote the smallest integer k such that for all $0 \leq t < k$, $A \notin Y_t$ and for all $t \geq k$, $A \in Y_t$. (Notice that the existence of $\rho(A)$ is guaranteed because T_{π_1} is a monotonic, fix-point operator.)

We prove (*) by induction over $\rho(trans(n_q, t_1, t_2))$.

Base: $\rho(trans(n_q, t_1, t_2)) = 0$. Then π_1 contains a rule of the form $trans(n_q, t_1, t_2) \leftarrow$. Because $q \neq \mathbf{null}$, we know that $trans(n_q, t_1, t_2) \leftarrow$ comes from a rule r of the form (109), (110), or (116).

- r is of the form (109). So, q is some action a, i.e., $action(a)$ and $occ(a, t)$ both belong to X. Further, $t_2 = t_1 + 1$. Because of Corollary 5.1 we know that a is executable in $s_{t_1}(X)$ and $s_{t_2}(X) \in \Phi(a, s_{t_1}(X))$. Since $s_t(M) = s_t(X)$ for every t, we have that $s_{t_1}(M)\ a\ s_{t_2}(M)$ is a trace of q.
- r is of the form (110). Then $q = \phi, t_2 = t_1 = t$, where ϕ is a formula and $hf(n_\phi, t)$ is in X. By Theorem 3, ϕ holds in $s_t(X)$. Again, because $s_t(M) = s_t(X)$, so $s_t(M)$ is a trace of q.
- r is of the form (116). Then, $t_1 = t_2$, $while(n_q, \phi, p_1) \in X$, and $hf(n_\phi, t_1) \notin X$. That is, q is the program "**while** ϕ **do** p_1" and ϕ does not holds in $s_{t_1}(M)$. Thus, $s_{t_1}(M)$ is a trace of q.

Step: Assume that we have proved (*) for $\rho(trans(n_q, t_1, t_2)) \leq k$. We need to prove it for the case $\rho(trans(n_q, t_1, t_2)) = k + 1$.

Because $trans(n_q, t_1, t_2)$ is in $T_{\pi_1}(Y_k)$, there is some rule $trans(n_q, t_1, t_2) \leftarrow A_1, \ldots A_m$ in π_1 such that all $A_1, \ldots A_m$ are in Y_k. From the construction of π_1, we have the following cases:

- r is a rule of the form (111). Then, there exists q_1, q_2, t' such that $proc(n_q, n_{q_1}, n_{q_2}) \in X$, and $trans(n_{q_1}, t_1, t') \in Y_k$ and $trans(n_{q_2}, t', t_2)$. Hence, $\rho(trans(n_{q_1}, t_1, t')) \leq k$ and $\rho(trans(n_{q_2}, t', t_2)) \leq k$. By inductive hypothesis, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t'-1}s_{t'}(M)$ is a trace of q_1 and $s_{t'}(M)a_{t'}s_{t'+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_2. Since $proc(n_q, nq_1, n_{q_2}) \in X$ we know that $q = q_1; q_2$. By definition, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q.
- r is a rule of the form (112). Then, $choiceAction(n_q)$ is in X. So, q is a choice program, say $q = q_1 \mid q_2 \ldots \mid q_l$. In addition, there exists $1 \leq j \leq l$ such that $in(n_{q_j}, n_q) \in X$ and $trans(n_{q_j}, t_1, t_2) \in Y_k$. By the definition of ρ, $\rho(trans(q_j, t_1, t_2)) \leq k$. By inductive hypothesis, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_j. By Definition 5, it is also a trace of q.
- r is a rule of the form (113). Then, by the construction of π_1, there exists ϕ, q_1, q_2 such that $if(n_q, n_\phi, n_{q_1}, n_{q_2}) \in X$, $hf(n_\phi, t_1) \in X$, and $trans(n_{q_1}, t_1, t_2) \in Y_k$. Thus q is the program "**if** ϕ **then** q_1 **else** q_2" and $\rho(trans(n_{q_1}, t_1, t_2)) \leq k$. Again, by inductive hypothesis, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_1. Because of Theorem 3, ϕ holds in $s_{t_1}(M)$. Hence, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q.
- r is a rule of the form (114). Similarly to the above, there exist $if(n_q, n_\phi, n_{q_1}, n_{q_2}) \in X$, $hf(n_\phi, t_1) \notin X$, and $trans(n_{q_2}, t_1, t_2) \in Y_k$. This means that $\rho(trans(n_{q_2}, t_1, t_2)) \leq k$. Hence, by inductive hypothesis and Theorem 3, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_2 and ϕ is false in $s_{t_1}(M)$, which mean that $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of "**if** ϕ **then** q_1 **else** q_2", i.e., a trace of q.

- r is a rule of the form (115). This implies that there exist a formula ϕ, a program q_1 and a time point $t' > t_1$ such that $while(n_q, n_\phi, n_{q_1}) \in X$ and $hf(n_\phi, t_1) \in X$, $trans(n_{q_1}, t_1, t')$ and $trans(n_q, t', t_2)$ are in Y_k. It follows that q is the program "**while** ϕ **do** q_1". Furthermore, ϕ holds in $s_{t_1}(M)$, and $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t'-1}s_{t'}(M)$ is a trace of q_1 and $s_{t'}(M)a_{t'}s_{t'+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q. By Definition 5, this implies that $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q.

- r a rule of the form is (117). Then, $choiceArgs(n_q)$ is in X and q has the form **pick**$(\vec{x}, f(\vec{x}), q_1(\vec{x}))$. $trans(n_q, t_1, t_2) \in Y$ implies that there exists a $\vec{x}_c$ such that $hf(n_{f(\vec{x}_c)}, t_1) \in X$ and $trans(n_{q_1(\vec{x}_c)}, t_1, t_2) \in Y_k$. By the definition of ρ, $\rho(trans(n_{q_1}, t_1, t_2)) \leq k$. By induction, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ is a trace of program $q_1(\vec{x}_c)$. Together with the fact that $f(\vec{x}_c)$ holds in s_{t_1}, we conclude that $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q.

The above cases prove the inductive step for (*). The lemma follows immediately since $trans(n_p, 0, n)$ belongs to M. $\square$

To prove the reverse of Lemma 5, we define a function μ that maps each program q into an integer $\mu(q)$ that reflects the complexity of q (or the number of nested operators in q). $\mu(q)$ is defined recursively over the construction of q as follows.

- For $q = \phi$ and ϕ is a formula, or $q = a$ and a is an action, $\mu(q) = 0$.
- For $q = q_1; q_2$ or $q =$ **if** ϕ **then** q_1 **else** q_2, $\mu(q) = 1 + \mu(q_1) + \mu(q_2)$.
- For $q = q_1 \mid \ldots \mid q_m$, $\mu(q) = 1 + \max\{\mu(q_i) \mid i = 1, \ldots, m\}$.
- For $q =$ **while** ϕ **do** q_1, $\mu(q) = 1 + \mu(q_1)$.
- For $q =$ **pick**$(\vec{x}, f(\vec{x}), q(\vec{x}))$, $\mu(q) = 1 + \max\{\mu(q(\vec{x}_c)) \mid \vec{x}_c$ is a ground instantiation of $\vec{x}\}$.

It is worth noting that $\mu(q)$ is always defined for well-defined programs.

Lemma 6 *Let (D, Γ) be a consistent action theory, p be a program, and $s_0a_0\ldots s_{n-1}a_n$ be a trace of p. Then Π_n^T has an answer set M such that*

- *$occ(a_i, t) \in M$ for $0 \leq i \leq n-1$,*
- *$s_t = s_t(M)$ for every $0 \leq t \leq n$, and*
- *$trans(n_p, 0, n) \in M$.*

Proof. We prove the lemma by constructing an answer set of Π_n^T that satisfies the conditions of the lemma. Similar to the proof of Lemma 5, we split Π_n^T using $U = lit(\pi) \cup r(p)$. Further, M is an answer set of Π_n^T iff $M = X \cup Y$ where X is an answer set of $b_U(\Pi_n^T)$ and Y is an answer set of $\pi_1 = e_U(\Pi_n^T \setminus b_U(\Pi_n^T), X)$, which is the program consisting of the rules (109)-(118) with the corresponding conditions.

Because $s_0a_0\ldots a_{n-1}s_n$ is a trace of p, it is a trajectory in D. By Corollary 5.2, we know that π has an answer set X' that satisfies the two conditions:

- $occ(a_i, t) \in X'$ for $0 \leq i \leq n-1$ and
- $s_t = s_t(X')$ for every $0 \leq t \leq n$.

Because $r(p)$ consists of only rules and atoms encoding the program p, it is easy to see that there exists an answer set X of $\pi \cup r(p)$ such that $X' \subseteq X$. Clearly, X also satisfies the two conditions:

- $occ(a_i, t) \in X$ for $0 \leq i \leq n-1$ and
- $s_t = s_t(X)$ for every $0 \leq t \leq n$.

Since π_1 is a positive program we know that π_1 has a unique answer set, say Y. From the splitting theorem, we have that $M = X \cup Y$ is an answer set of Π_n^T. Because $s_t(X) = s_t(M)$, M satisfies the first two conditions of the lemma. It remains to be shown that M also satisfies the third condition of the lemma. We prove this by proving a stronger conclusion:

(*) If q is a program occurring in p, and there exists two integers t_1 and t_2 such that $s_{t_1}(M)a_{t_1} \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q then $trans(n_q, t_1, t_2) \in M$. (the states $s_i(M) = s_i$ – see above – and the actions a_i are defined as in the Lemma's statement)

We prove (*) by induction over $\mu(q)$, the complexity of the program q.

Base: $\mu(q) = 0$. There are only two cases:

- $q = \phi$ for some formula ϕ, and hence, by Definition 5, we have that $t_2 - t_1 = 0$. It follows from the assumption that $s_{t_1}(M)$ is a trace of q that $s_{t_1}(M)$ satisfies ϕ. By Theorem 3, $hf(n_\phi, t_1) \in X$, and hence, we have that $trans(n_\phi, t_1, t_1) \in Y$ (because of rule (110)).
- $q = a$ where a is an action. Again, by Definition 5, we have that $t_2 = t_1 + 1$. From the assumption that $s_{t_1}(M)a_{t_1}s_{t_2}(M)$ is a trace of q we have that $a_{t_1} = a$. Thus, $occ(a, t_1) \in M$. By rule (109) of π_1, we conclude that $trans(a, t_1, t_2) \in Y$, and thus, $trans(a, t_1, t_2) \in M$.

The above two cases prove the base case.

Step: Assume that we have proved (*) for every program q with $\mu(q) \leq k$. We need to prove it for the case $\mu(q) = k + 1$. Because $\mu(q) > 0$, we have the following cases:

- $q = q_1; q_2$. By Definition 5, there exists t', $t_1 \leq t' \leq t_2$, such that $s_{t_1}a_{t_1} \ldots s_{t'}$ is a trace of q_1 and $s_{t'}a_{t'} \ldots s_{t_2}$ is a trace of q_2. Because $\mu(q_1) < \mu(q)$ and $\mu(q_2) < \mu(q)$, by inductive hypothesis, we have that $trans(n_{q_1}, t_1, t') \in M$ and $trans(n_{q_2}, t', t_2) \in M$. $q = q_1; q_2$ implies $proc(n_q, n_{q_1}, n_{q_2}) \in M$. By rule (111), $trans(n_q, t_1, t_2)$ must be in M.
- $q = q_1 \mid \ldots \mid q_i$. Again, by Definition 5, $s_{t_1}a_{t_1} \ldots a_{t_2-1}s_{t_2}$ is a trace of some q_j. Since $\mu(q_j) < \mu(q)$, by inductive hypothesis, we have that $trans(n_{q_j}, t_1, t_2) \in M$. Because of rule (112), $trans(n_q, t_1, t_2)$ is in M.
- $q =$ **if** ϕ **then** q_1 **else** q_2. Consider two cases:
  - ϕ holds in s_{t_1}. This implies that $s_{t_1}(M)a_{t_1} \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_1. Because of Theorem 3, $hf(n_\phi, t_1) \in M$. Since $\mu(q_1) < \mu(q)$, $trans(n_{q_1}, t_1, t_2) \in M$ by inductive hypothesis. Thus, according to rule (113), $trans(n_q, t_1, t_2)$ must belong to M.
  - ϕ does not holds in s_{t_1}. This implies that $s_{t_1}(M)a_{t_1} \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q_2. Because of Theorem 3, $hf(n_\phi, t_1)$ does not hold in M. Since $\mu(q_1) < \mu(q)$, $trans(n_{q_2}, t_1, t_2)$ is in M by inductive hypothesis. Thus, according to rule (114), $trans(n_q, t_1, t_2) \in M$.
- $q =$ **while** ϕ **do** q_1. We prove this case by induction over the length of the trace, $t_2 - t_1$.
  - **Base:** $t_2 - t_1 = 0$. This happens only when ϕ does not hold in $s_{t_1}(M)$. As such, because of rule (116), $trans(n_q, t_1, t_2)$ is in M. The base case is proved.

– **Step:** Assume that we have proved the conclusion for this case for $0 \leq t_2 - t_1 < l$. We will show that it is also correct for $t_2 - t_1 = l$. Since $t_2 - t_1 > 0$, we conclude that ϕ holds in s_{t_1} and there exists $t_1 < t' \leq t_2$ such that $s_{t_1} a_{t_1} \ldots s_{t'}$ is a trace of q_1 and $s_{t'} a_{t'} \ldots s_{t_2}$ is a trace of q. We have $\mu(q_1) < \mu(q)$, $t' - t_1 \leq t_2 - t_1$ and $t_2 - t' < t_2 - t_1 = l$. By inductive hypothesis, $trans(n_{q_1}, t_1, t')$ and $trans(n_q, t', t_2)$ are in M. By Theorem 3, $hf(n_\phi, t_1)$ is in M and from the rule (115), $trans(n_q, t_1, t_2)$ is in M.

- $q =$ $\mathbf{pick}(\vec{x}, f(\vec{x}), q_1(\vec{x}))$. So, there exists $\vec{x}_c$, such that $f(\vec{x}_c)$ holds in s_{t_1} and the trace of q is a trace of $q_1(\vec{x}_c)$. Since $\mu(q_1(\vec{x}_c)) < \mu(q)$, we have that $trans(n_{q_1(\vec{x}_c)}, t_1, t_2) \in M$. This, together with the fact that $choiceArgs(n_q, n_{f(\vec{x}_c)}, n_{q_1(\vec{x}_c)}) \in r(p)$ and $hf(n_\phi, t_1) \in M$ (Theorem 3), and the rule (117) imply that $trans(n_q, t_1, t_2)$ is in M.

The above cases prove the inductive step of (*). The conclusion of the lemma follows. □

We now prove the Theorem 6.

Theorem 6 Let (D, Γ) be a consistent action theory and p be a program. Then,

(i) for every answer set M of Π_n^T with $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$, $s_0(M)a_0 \ldots a_{n-1}s_n(M)$ is a trace of p; and

(ii) if $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p then there exists an answer set M of Π_n^T such that $s_j = s_j(M)$ and $occ(a_i, i) \in M$ for $j \in \{0, \ldots, n\}$ and $i \in \{0, \ldots, n-1\}$.

Proof. (i) follows from Lemma 5 and (ii) follows from Lemma 6. □

Appendix A.4 - Proof of Theorem 7

Let p now be a general program. To prove Theorem 7, we will extend the Lemmas 5-6 to account for general programs. Similarly to the proofs of Lemmas 5-6, we will split Π_n^{HTN} by the set $U = lit(\pi) \cup r(p)$. Thus M is an answer set of Π_n^{HTN} iff $M = X \cup Y$ where X is an answer set of $\pi \cup r(p)$ and Y is an answer set of the program $e_U(\Pi_n^{HTN} \setminus b_U(\Pi_n^{HTN}), X)$ which consists of the rules of program π_1 (with the difference that a program is now a general program) and the program π_2 which consists of the following rules:

$$
\begin{array}{rcll}
trans(N,T_1,T_2) & \leftarrow & not\ nok(N,T_1,T_2). & (119)\\
 & & (\text{if } htn(N,S,C) \in X) & \\
1\{begin(N,I,T_3,T_1,T_2) : between(T_3,T_1,T_2)\}1 & \leftarrow & trans(N,T_1,T_2). & (120)\\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X) & \\
1\{end(N,I,T_3,T_1,T_2) : between(T_3,T_1,T_2)\}1 & \leftarrow & trans(N,T_1,T_2). & (121)\\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X) & \\
used(N,T,T_1,T_2) & \leftarrow & begin(N,I,B,T_1,T_2), end(N,I,E,T_1,T_2), & (122)\\
 & & B \leq T \leq E. & \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X) & \\
not\_used(N,T,T_1,T_2) & \leftarrow & not\ used(N,T,T_1,T_2). & (123)\\
overlap(N,T,T_1,T_2) & \leftarrow & begin(N,I_1,B_1,T_1,T_2), end(N,I_1,E_1,T_1,T_2), & (124)\\
 & & begin(N,I_2,B_2,T_1,T_2), end(N,I_2,E_2,T_1,T_2), & \\
 & & B_1 \leq T \leq E_1, B_2 < T < E_2. &
\end{array}
$$

$$
\begin{array}{rcll}
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X,\ in(I_2,S) \in X) & \\
nok(N,T_1,T_2) & \leftarrow & T_3 > T_4, begin(N,I,T_3,T_1,T_2), end(N,I,T_4,T_1,T_2). & (125) \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X) & \\
nok(N,T_1,T_2) & \leftarrow & T_3 \leq T_4, begin(N,I,T_3,T_1,T_2), end(N,I,T_4,T_1,T_2), & (126) \\
 & & not\ trans(I,T_3,T_4). & \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X) & \\
nok(N,T_1,T_2) & \leftarrow & T_1 \leq T \leq T_2, not\_used(N,T,T_1,T_2). & (127) \\
 & & (\text{if } htn(N,S,C) \in X) & \\
nok(N,T_1,T_2) & \leftarrow & T_1 \leq T \leq T_2, overlap(N,T,T_1,T_2). & (128) \\
 & & (\text{if } htn(N,S,C) \in X) & \\
nok(N,T_1,T_2) & \leftarrow & begin(N,I_1,B_1,T_1,T_2), & (129) \\
 & & begin(N,I_2,B_2,T_1,T_2), & \\
 & & B_1 > B_2. & \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I_1,S) \in X,\ in(I_2,S) \in X, & \\
 & & in(O,C) \in X,\ order(O,I_1,I_2) \in X)) & \\
nok(N,T_1,T_2) & \leftarrow & end(N,I_1,E_1,T_1,T_2), & (130) \\
 & & begin(N,I_2,B_2,T_1,T_2), E_1 < T_3 < B_2. & \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I_1,S) \in X,\ in(I_2,S) \in X, & \\
 & & in(O,C)\ \in X,\ maintain(O,F,I_1,I_2) \in X, & \\
 & & \text{and } hf(F,T_3) \notin X) & \\
nok(N,T_1,T_2) & \leftarrow & begin(N,I,B,T_1,T_2), end(N,I,E,T_1,T_2), & (131) \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X, & \\
 & & in(O,C) \in X, precondition(O,F,I) \in X & \\
 & & \text{and } hf(F,B) \notin X) & \\
nok(N,T_1,T_2) & \leftarrow & begin(N,I,B,T_1,T_2), end(N,I,E,T_1,T_2). & (132) \\
 & & (\text{if } htn(N,S,C) \in X,\ in(I,S) \in X, & \\
 & & in(O,C) \in X,\ postcondition(O,F,I) \in X, & \\
 & & \text{and } hf(F,E) \notin X) &
\end{array}
$$

Let π_2 be the program consisting of the above set of rules. Thus $e_U(\Pi_n^{HTN} \setminus b_U(\Pi_n^{HTN}), X) = \pi_1 \cup \pi_2$. We will continue to use the complexity of program defined in the last appendix and extend it to allow the HTN-construct by adding the following to the definition of $\mu(q)$.

- For $q = (S, C)$, $\mu(q) = 1 + \Sigma_{p \in S} \mu(p)$.

Notice that every literal of the program $\pi_1 \cup \pi_2$ has the first parameter as a program[17]. Hence, we can associate $\mu(q)$ to each literal u of π' where q is the first parameter of u. For instance, $\mu(trans(q,t_1,t_2)) = \mu(q)$ or $\mu(nok(q,t_1,t_2)) = \mu(q)$ etc.. Since we will continue using splitting theorem in our proofs, the following observation is useful.

Observation 5.2 *The two cardinality constraint rules (54) and (55) can be replaced by the following normal logic program rules:*

$$
\begin{array}{rcl}
begin(N,I,T,T_1,T_2) & \leftarrow & htn(S,C), in(I,S), trans(N,T_1,T_2), \\
 & & T_1 \leq T \leq T_3 \leq T_2, not\ nbegin(N,I,T,T_1,T_2). \\
nbegin(N,I,T,T_1,T_2) & \leftarrow & htn(S,C), in(I,S), trans(N,T_1,T_2),
\end{array}
$$

[17] More precisely, a program name.

$$
\begin{array}{rcl}
 & & T_1 \leq T \leq T_2, T_1 \leq T_3 \leq T_2, T \neq T_3, begin(N, I, T_3, T_1, T_2). \\
occur(N, I, T_1, T_2) & \leftarrow & htn(S, C), in(I, S), T_1 \leq T \leq T_2, begin(N, I, T, T_1, T_2). \\
 & \leftarrow & htn(S, C), in(I, S), trans(N, T_1, T_2), not\ occur(N, I, T_1, T_2).
\end{array}
$$

That is, let π^* be the program obtained from $\pi_1 \cup \pi_2$ by replacing the rules (54)-(55) with the above set of rules. Then, M is an answer set of $\pi_1 \cup \pi_2$ iff $M' = M \cup \{occur(N, I, T_1, T_2) \mid begin(N, I, T, T_1, T_2) \in M$ for some $T_1 \leq T \leq T_2\} \cup \{nbegin(N, I, T', T_1, T_2) \mid T \neq T',\ T_1 \leq T, T' \leq T_2$ such that $begin(N, I, T, T_1, T_2) \in M\}$ is an answer set of π^*.

The next lemma generalizes Lemma 5.

Lemma 7 *Let q be a general program, Y be an answer set of the program $e_U(\Pi_n^{HTN} \setminus b_U(\Pi_n^{HTN}), X)$ (i.e. program $\pi_1 \cup \pi_2$), and t_1, t_2 be two time points such that $q \neq$ **null** and $trans(n_q, t_1, t_2) \in Y$. Then, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q where $M = X \cup Y$.*

Proof. Let $\pi' = \pi_1 \cup \pi_2$ and $U_k = \{u \mid u \in lit(\pi'),\ \mu(u) \leq k\}$.

From observation 5.2, we know that we can use the splitting theorem on π'. It is easy to see that $\langle U_k \rangle_k^\infty$ is a splitting sequence of π'. From the finiteness of π' and the splitting sequence theorem, we have that $Y = \bigcup_0^\infty Y_i$ where

1. Y_0 is an answer set of the program $b_{U_0}(\pi')$ and
2. for every integer i, Y_{i+1} is an answer set for $e_{U_i}(b_{U_{i+1}}(\pi') \setminus b_{U_i}(\pi'), \bigcup_{j \leq i} Y_j)$.

We prove the lemma by induction over $\mu(q)$.

Base: $\mu(q) = 0$. From $trans(n_q, t_1, t_2) \in Y$, we have that $trans(n_q, t_1, t_2) \in Y_0$. It is easy to see that $b_{U_0}(\pi')$ consists of all the rules of π_1 whose program has level 0. It follows from Lemma 5 $s_{t_1}(M)a_{t_1}s_{t_1+1}(M) \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q. The base case is proved.

Step: Assume that we have proved the lemma for $\mu(q) = k$. We prove it for $\mu(q) = k + 1$. From the fact that $trans(n_q, t_1, t_2) \in M$ and $\mu(n_q) = k + 1$ we have that $trans(n_q, t_1, t_2) \in Y_{k+1}$ where Y_{k+1} is an answer set of the program $e_{U_k}(b_{U_{k+1}}(\pi') \setminus b_{U_k}(\pi'), \bigcup_{j \leq k} Y_k)$ which consists of rules of the form (119)-(132) and (111)-(117) whose program has the level $k + 1$, i.e., $\mu(N) = k + 1$. Because $trans(n_q, t_1, t_2) \in Y$ we know that there exists a rule that supports $trans(n_q, t_1, t_2)$. Let r be such a rule. There are following cases:

- r is a rule of the form (111)-(117), the argument is similar to the argument using in the inductive step for the corresponding case in Lemma 5. Notice a minor difference though: in Lemma 5, we do not need to use $\mu(q)$.
- r is a rule of the form (119), which implies that $q = (S, C)$ for some set of programs S and set of constraints C. By definition of answer sets, we know that $nok(n_q, t_1, t_2) \notin Y_{k+1}$. Furthermore, because of the rules (120) and (121), the fact that $trans(n_q, t_1, t_2) \in Y_{k+1}$ and the definition of weight constraint rule, we conclude that for each $q_j \in S$ there exists two numbers j_b and j_e, $t_1 \leq j_b, j_e \leq t_2$ such that $begin(n_q, n_{q_j}, j_b, t_1, t_2) \in Y_{k+1}$ and $end(n_q, n_{q_j}, j_e, t_1, t_2) \in Y_{k+1}$. Because of rule (126), we conclude that $trans(n_{q_j}, j_b, j_e) \in \bigcup_{i \leq k} Y_i$. Otherwise, we have that $nok(n_q, t_1, t_2) \in Y_{k+1}$, and hence, $trans(n_q, t_1, t_2) \notin Y_{k+1}$, which is a contradiction. By definition of $\mu(q)$, we have that $\mu(q_j) < \mu(q)$. Thus, by inductive hypothesis, we can conclude that: for every $q_j \in S$, there exists two numbers j_b and j_e, $t_1 \leq j_b, j_e \leq t_2$, $s_{j_b}(M)a_{j_b} \ldots a_{j_e-1}s_{j_e}(M)$ is a trace of q_j.

Furthermore, rules (122)-(128) imply that the set $\{j_b \mid q_j \in S\}$ creates a permutation of $\{1, \ldots, |S|\}$ that satisfies the first condition of Definition 7.

Consider now an ordering $q_{j_1} < q_{j_2}$ in C. This implies that the body of rule (129) will be satisfied if $j_{b_1} > j_{b_2}$ which would lead to $trans(n_q, t_1, t_2) \notin Y_{k+1}$. Again, this is a contradiction. Hence, we must have $j_{b_1} \leq j_{b_2}$ that means that the permutation $\{j_b \mid q_j \in S\}$ also satisfies the second condition of Definition 7.

Similarly, using (130)-(132) we can prove that the permutation $\{j_b \mid q_j \in S\}$ also satisfies the third and fourth conditions of Definition 7.

It follows from the above arguments that $s_{t_1}(M)a_{t_1} \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q. The inductive step is proved for this case.

The above cases prove the inductive step. This concludes the lemma. □

In the next lemma, we generalize the Lemma 6.

Lemma 8 *Let (D, Γ) be a consistent action theory, p be a general program, and $s_0a_0 \ldots a_{n-1}s_n$ be a trace of p. Then, there is an answer set M of Π_n^{HTN} such that $s_i(M) = s_i$ and $occ(a_i, i) \in M$ and $trans(n_p, 0, n) \in M$.*

Proof. Based on our discussion on splitting Π_n^{HTN} using $lit(\pi) \cup r(q)$ and the fact that $s_0a_0 \ldots a_{n-1}s_n$ is also a trace in D, we know that there exists an answer set X of $\pi \cup r(p)$ such that $s_i(X) = s_i$ and $occ(a_i, i) \in X$. Thus, it remains to be shown that there exists an answer set Y of $\pi' = \pi_1 \cup \pi_2$ such that $trans(n_p, 0, n) \in Y$. Similar to the proof of Lemma 7, we use $\langle U_k \rangle_k^\infty$ as a splitting sequence of π' where $U_k = \{u \mid u \in lit(\pi'),\ \mu(u) \leq k\}$. From the splitting sequence theorem, we have that $Y = \bigcup_0^\infty Y_i$ where

1. Y_0 is an answer set of the program $b_{U_0}(\pi')$ and
2. for every integer i, Y_{i+1} is an answer set for $e_{U_i}(b_{U_{i+1}}(\pi') \setminus b_{U_i}(\pi'), \bigcup_{j \leq i} Y_j)$.

We prove the lemma by induction over $\mu(q)$. Similar to Lemma 6, we prove this by proving a stronger conclusion:

(*) There exists an answer set $Y = \bigcup_0^\infty Y_i$ of π' such that for every program $q \neq$ **null** occurring in p, $s_{t_1}a_{t_1} \ldots a_{t_2-1}s_{t_2}$ is a trace of q iff $trans(n_q, t_1, t_2) \in Y_{\mu(q)}$. (the states s_i and the actions a_i are defined as in the Lemma's statement)

We will prove (*) by induction over $\mu(q)$.

Base: $\mu(q) = 0$. Similar to the base case in Lemma 6 .

Step: Assume that we have proved (*) for $\mu(q) \leq k$. We need to prove (*) for $\mu(q) = k + 1$. We will construct an answer set of $\pi^+ = e_{U_k}(b_{U_{k+1}}(\pi') \setminus b_{U_k}(\pi'), \bigcup_{j \leq k} Y_k)$ such that for every program q occurring in p with $\mu(q) = k + 1$, if $s_{t_1}a_{t_1} \ldots a_{t_2-1}s_{t_2}$ is a trace of q then $trans(n_q, t_1, t_2) \in Y_{k+1}$.

Let Y_{k+1} be the set of atoms defined as follows.

- For every program q with $\mu(q) = k + 1$, if q is not of the form (S, C) and $s_{t_1}a_{t_1} \ldots a_{t_2-1}s_{t_2}$ is a trace of q, Y_{k+1} contains $trans(n_q, t_1, t_2)$.

- For every program q with $\mu(q) = k+1$, $q = (S, C)$, and $s_{t_1} a_{t_1} \ldots a_{t_2-1} s_{t_2}$ is a trace of q. By definition, there exists a permutation $\{j_1, \ldots, j_{|S|}\}$ of $\{1, \ldots, |S|\}$ satisfying the conditions (a)-(d) of Item 8 (Definition 7). Consider such a permutation. To simplify the notation, let us denote the begin- and end-time of a program $q_j \in S$ in the trace of q by b_j and e_j, respectively, i.e., $s_{b_j} a_{b_j} \ldots s_{e_j}$ is a trace of q_j. Then, Y_{k+1} contains $trans(n_q, t_1, t_2)$ and the following atoms:

  1. $begin(n_q, n_{q_j}, b_j, t_1, t_2)$ for every $q_j \in S$,
  2. $end(n_q, n_{q_j}, e_j, t_1, t_2)$ for every $q_j \in S$, and
  3. $used(n_q, t, t_1, t_2)$ for for every $q_j \in S$ and $b_j \leq t \leq e_j$.

- Y_{k+1} does not contain any other atoms except those mentioned above.

It is easy to see that Y_{k+1} satisfies (*) for every program q with $\mu(q) = k+1$. Thus, we need to show that Y_{k+1} is indeed an answer set of π^+. First, we prove that Y_{k+1} is closed under $(\pi^+)^Y_{k+1}$. We consider the following cases:

- r is a rule of the form (111). Obviously, if r belongs to $(\pi^+)^Y_{k+1}$, then $q = q_1; q_2$ and there exists a $t_1 \leq t' \leq t_2$ such that $trans(n_{q_1}, t_1, t')$ and $trans(n_{q_2}, t', t_2)$ belong to $\bigcup_{j \leq k} Y_k$ because $\mu(q_1) < \mu(q)$ and $\mu(q_2) < \mu(q)$. By inductive hypothesis, $s_{t_1} a_{t_1} \ldots s_{t'}$ is a trace of q_1 and $s_{t'} a_{t'} \ldots s_{t_2}$ is a trace of q_2. By Definition 5, $s_{t_1} a_{t_1} \ldots s_{t_2}$ is a trace of q. By construction of Y_{k+1} we have that $trans(n_q, t_1, t_2) \in Y_{k+1}$. This shows that Y_{k+1} is closed under r. Similar arguments conclude that Y_{k+1} is closed under the rule of the form (112)-(117).

- r is a rule of the form (119) of $(\pi^+)^Y_{k+1}$. Then, $q = (S, C)$ and by construction of Y_{k+1}, if $s_{t_1} a_{t_1} \ldots s_{t_2}$ is a trace of q then we have $trans(n_q, t_1, t_2) \in Y_{k+1}$. Thus. Y_{k+1} is closed under the rules of the form (119) too.

- r is a rule of the form (120) and (121). Y_{k+1} is also closed under r because whenever $trans(n_q, t_1, t_2) \in Y_{k+1}$, we now that there is a trace $s_{t_1} a_{t_1} \ldots s_{t_2}$ of q, and hence, by Definition 7, we conclude the existence of the begin- and end-time points b_j and e_j of q_j, respectively. By construction of Y_{k+1}, we have that $begin(n_q, n_{q_j}, b_j, t_1, t_2)$ and $end(n_q, n_{q_j}, e_j, t_1, t_2)$ belong to Y_{k+1} and for each q_j, there is a unique atom of this form in Y_{k+1}. Hence, Y_{k+1} is closed under rules of the form (120) and (121).

- r is a rule of the form (123)-(132). The construction of Y_{k+1} ensures that the body of r is not satisfied by Y_{k+1}, and hence, Y_{k+1} is closed under r.

- r is a rule of the form (122). Because $used(n_q, t, t_1, t_2)$ belongs to Y_{k+1} for every t, $t_1 \leq t \leq t_2$. we have that Y_{k+1} is closed under r too.

The conclusion that Y_{k+1} is closed under $(\pi^+)^Y_{k+1}$ follows from the above cases.

To complete the proof, we need to show that Y_{k+1} is minimal. Assume the contrary, there exists a proper subset Y' of Y_{k+1} such that Y' is closed under $(\pi^+)^{Y_{k+1}}$. Let $u \in Y_{k+1} \setminus Y'$. Since $u \in Y_{k+1}$, we have the following cases:

- u is the head of a rule of the form (111)-(117). By definition of π^+, we know that a rule of this form belongs to π^+ iff its body is empty. Thus, from the closeness of Y' we have that $u \in Y'$. This contradicts the fact that $u \notin Y'$.

- u is the head of a rule of the form (119). Similar to the above case, we can conclude that $u \in Y'$ which again contradicts the fact that $u \notin Y'$.

- u is the head of a rule r of the form (120). Because of $u \in Y_{k+1}$ we conclude that $trans(n_q, t_1, t_2) \in Y_{k+1}$. The above case concludes that $trans(n_q, t_1, t_2) \in Y'$. Since the body of r is true, we conclude that there exists some $q_j \in S$ such that Y' does not contain an atom of the form $begin(n_q, n_{q_j}, b_j, t_1, t_2)$. Thus, Y' is not closed under r. This contradicts the assumption that Y' is closed under $(\pi^+)^{Y_{k+1}}$.

- u is the head of a rule r of the form (121). Similar to the above case, we can prove that it violates the assumption that Y' is closed under $(\pi^+)^{Y_{k+1}}$.

- u is the head of a rule r of the form (122). Because $u \in Y_{k+1}$ we know that the body of r is satisfied by Y_{k+1}, and hence, r belongs to $(\pi^+)^{Y_{k+1}}$. Again, because of the closeness of Y', we conclude that $u \in Y'$ which violates the assumption that $u \notin Y'$.

The above cases imply that Y' is not closed under $(\pi^+)^{Y_{k+1}}$. Thus, our assumption that Y_{k+1} is not minimal is incorrect. Together with the closeness of Y_{k+1}, we have that Y_{k+1} is indeed an answer set of π^+. The inductive step is proved since Y_{k+1} satisfies (*) for every program q with $\mu(q) = k + 1$. This proves the lemma. $\square$

Theorem 7 Let (D, Γ) be a consistent action theory and p be a general program. Then,

(i) for every answer set M of Π_n^{HTN} with $occ(a_i, i) \in M$ for $i \in \{0, \ldots, n-1\}$, $s_0(M)a_0 \ldots a_{n-1}s_n(M)$ is a trace of p; and

(ii) if $s_0a_0 \ldots a_{n-1}s_n$ is a trace of p then there exists an answer set M of Π_n^{HTN} such that $s_j = s_j(M)$ and $occ(a_i, i) \in M$ for $j \in \{0, \ldots, n\}$ and $i \in \{0, \ldots, n-1\}$ and $trans(n_p, 0, n) \in M$.

Proof. (i) follows from Lemma 7 and (ii) follows from Lemma 8. $\square$

Appendix B - Splitting Theorem

Let r be a rule

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, a_n.$$

By $head(r)$, $body(r)$, and $lit(r)$ we denote a_0, $\{a_1, \ldots, a_n\}$, and $\{a_0, a_1, \ldots, a_n\}$, respectively. $pos(r)$ and $neg(r)$ denote the set $\{a_1, \ldots, a_m\}$ and $\{a_{m+1}, \ldots, a_n\}$, respectively.

For a program Π over the language $\mathcal{LP}$, a set of literals of $\mathcal{LP}$, A, is a splitting set of Π if for every rule $r \in \Pi$, r is of the form if $head(r) \in A$ then $lit(r) \subseteq A$.

Let A be a splitting set of Π. The *bottom of* Π *relative to* A, denoted by $b_A(\Pi)$, is the program consisting of all rules $r \in \Pi$ such that the head of r belongs to A.

Given a splitting set A for Π, and a set X of literals from $lit(b_A(\Pi))$, the *partial evaluation of* Π *by* X *with respect to* A, denoted by $e_A(\Pi, X)$, is the program obtained from Π as follows. For each rule $r \in \Pi \setminus b_A(\Pi)$ such that

1. $pos(r) \cap A \subseteq X$;
2. $neg(r) \cap A$ is disjoint from X;

there is a rule r' in $e_A(\Pi, X)$ such that

1. $head(r') = head(r)$, and
2. $pos(r') = pos(r) \setminus A$,
3. $neg(r') = neg(r) \setminus A$.

Let A be a splitting set of Π. A *solution to* Π *with respect to* A is a pair $\langle X, Y \rangle$ of set of literals satisfying the following two properties:

1. X is an answer set of $b_A(\Pi)$;
2. Y is an answer set of $e_A(\Pi \setminus b_A(\Pi), X)$;
3. $X \cup Y$ is consistent.

The splitting set theorem is as follows.

Theorem 8 (Splitting Set Theorem, [27]) *Let A be a splitting set for a program Π. A set A of literals is a consistent answer set of Π iff $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π with respect to A.* $\square$

A *sequence* is a family whose index set is an initial segment of ordinals $\{\alpha \mid \alpha < \mu\}$. A sequence $\langle A_\alpha \rangle_{\alpha<\mu}$ of sets is *monotone* if $A_\alpha \subseteq A_\beta$ whenever $\alpha < \beta$, and *continuous* if, for each limit ordinal $\alpha < \mu$, $A_\alpha = \bigcup_{\gamma<\alpha} A_\gamma$.

A *splitting sequence* for a program Π is a nonempty, monotone, and continuous sequence $\langle A_\alpha \rangle_{\alpha<\mu}$ of splitting sets of Π such that $lit(\Pi) = \bigcup_{\alpha<\mu} A_\alpha$.

Let $\langle A_\alpha \rangle_{\alpha<\mu}$ be a splitting sequence of the program Π. A *solution to* Π *with respect to* A is a sequence $\langle E_\alpha \rangle_{\alpha<\mu}$ of set of literals satisfying the following conditions.

1. E_0 is an answer set of the program $b_{A_0}(\Pi)$;
2. for any α such that $\alpha + 1 < \mu$, $E_{\alpha+1}$ is an answer set for $e_{A_\alpha}(b_{A_{\alpha+1}}(\Pi) \setminus b_{A_\alpha}(\Pi), \bigcup_{\gamma\leq\alpha} E_\gamma)$;
3. For any limit ordinal $\alpha < \mu$, $E_\alpha = \emptyset$;
4. $\bigcup_{\gamma\leq\mu} E_\gamma$ is consistent.

The splitting set theorem is generalized for splitting sequence next.

Theorem 9 (Splitting Sequence Theorem, [27]) *Let $A = \langle A_\alpha \rangle_{\alpha<\mu}$ be a splitting sequence of the program Π. A set of literals E is a consistent answer set of Π iff $E = \bigcup_{\alpha<\mu} E_\alpha$ for some solution $\langle E_\alpha \rangle_{\alpha<\mu}$ to Π with respect to A.* $\square$

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. In *The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems*, 2000.

[3] C. Baral. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 2017–2023. Morgan Kaufmann Publishers, San Francisco, CA, 1995.

[4] C. Baral. *Knowledge Representation, reasoning, and declarative problem solving with Answer sets (Book draft)*. 2001.

[5] C. Baral and T. C. Son. Extending ConGolog to allow partial ordering. In *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL), LNCS, Vol. 1757*, pages 188–204, 1999.

[6] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[7] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence - Special issue on Heuristic Search*, 129(1-2):5–33, 2001.

[8] W. Burgard, A. B. Cremers, D. Fox, D. H́ahnel, G. Lakemeyer, Schulz D., W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 11–18. AAAI Press, 1998.

[9] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:161–204, 1994.

[10] 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.

[11] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, September 2001.

[12] G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[13] 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.

[14] P. Doherty and J. Kvarnstom. TALplanner: An Empirical Investigation of a Temporal Logic-based Forward Chaining Planner. In *Proceedings of the 6th Int'l Workshop on the Temporal Representation and Reasoning, Orlando, Fl. (TIME'99)*, 1999.

[15] 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.

[16] R. Fikes and N. Nilson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[17] M. Gelfond. Posting on TAG-mailing list, 1999.

[18] M. Gelfond and N. Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[19] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conf. and Symp.*, pages 1070–1080, 1988.

[20] M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 3(6), 1998.

[21] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligent Research*, 14:253–302, 2001.

[22] Y.C. Huang, B. Selman, and H. Kautz. Control knowledge in planning: Benefits and tradeoffs. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 511–517. AAAI Press, 1999.

[23] 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, Pittsburgh*, 1998.

[24] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning and Scheduling Systems*, 1998.

[25] 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, April-June 1997.

[26] V. Lifschitz. Answer set planning. In *Proceedings of ICLP*, pages 23–37, 1999.

[27] V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conf. on Logic Programming*, pages 23–38, 1994.

[28] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 92–106, 1999.

[29] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, October 1994.

[30] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398, 1999.

[31] N. McCain and M. Turner. A causal theory of ramifications and qualifications. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1978–1984. Morgan Kaufmann Publishers, San Mateo, CA, 95.

[32] S. McIlraith. Modeling and programming devices and Web agents. In *Proc. of the NASA Goddard Workshop on Formal Approaches to Agent-Based Systems, LNCS*. Springer-Verlag, 2000.

[33] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Conference on Artificial Intelligence*, pages 968–973. AAAI Press, 1999.

[34] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[35] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings ICLP & LPNMR*, pages 420–429, 1997.

[36] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics for weight constraint rules. In *Proceedings of the 5th International Conference on on Logic Programming and Nonmonotonic Reasoning*, pages 315–332, 1999.

[37] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[38] R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Proc. of the Second International Cognitive Robotics Workshop, Berlin*, 2000.

[39] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[40] K. Sagonas, T. Swift, and D.S. Warren. Xsb as an efficient deductive database engine. In *Proceedings of the SIGMOD*, pages 442 – 453, 1994.

[41] V.S. Subrahmanian and C. Zaniolo. Relating stable models and ai planning domains. In *Proceedings of the International Conference on Logic Programming*, pages 233–247, 1995.

[42] L. Tuan and C. Baral. Effect of knowledge representation on model based planning : experiments using logic programming encodings. In *Proc. of AAAI Spring symposium on "Answer Set Programming:Towards Efficient and Scalable Knowledge Representation and Reasoning"*, pages 110–115, 2001.

[43] H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.

[44] D. Wilkins and M. desJardines. A call for knowledge-based planning. *AI Magazine*, 22(1):99–115, Spring 2001.