Domain-Dependent Knowledge in Answer Set Planning

TRAN CAO SON
New Mexico State University
CHITTA BARAL and NAM TRAN
Arizona State University
and
SHEILA MCILRAITH
University of Toronto

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 (AnsProlog*). AnsProlog* is designed to plan without control knowledge. We show how temporal, procedural and HTN-based control knowledge can be incorporated into AnsProlog* by the modular addition of a small number of domain-dependent rules, without the need to modify the planner. We formally prove the correctness of our planner, both in the absence and presence of the control knowledge. Finally, we perform some initial experimentation that demonstrates the potential reduction in planning time that can be achieved when procedural domain knowledge is used to solve planning problems with large plan length.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Plan generation*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic programming*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representation languages*

General Terms: Planning, Control Knowledge, Answer Set Planning

Additional Key Words and Phrases: Reasoning about actions, Procedural knowledge

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

Author's address: T. C. Son, Computer Science Department, PO Box 30001, MSC CS, New Mexico State University, Las Cruces, NM 88003, USA.
C. Baral and N. Tran, Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, USA.
S. McIlraith, Department of Computer Science, University of Toronto, Toronto, Canada M5S 5H3.

[Fikes and Nilson 1971]) of the effects 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 undecidable in the general case [Chapman 1987; Erol et al. 1995]. It reduces to PSPACE-complete for finite and deterministic domains [Bylander 1994]. By making certain assumptions such as fixing the length of plans, and requiring actions to be deterministic the complexity reduces to NP-complete.

The ability to plan is widely recognized to be an important characteristic of an intelligent entity. Thus, when developing intelligent systems, we often need to incorporate planning capabilities, despite their inherent complexity. Since the complexity is due to the exponential size of the search space, planning approaches that overcome this complexity require efficient and intelligent search. This is at the crux of three common and successful approaches to planning: (i) using heuristics [Bonet and Geffner 2001; Hoffmann and Nebel 2001; Blum and Furst 1997] that are derived from the problem description, (ii) translating the planning problem into a model finding problem in a suitable logic and using model finding techniques for that logic [Kautz and Selman 1998a], and (iii) using domain-dependent control knowledge[1] [Bacchus and Kabanza 2000; Doherty and Kvarnstom 1999; Nau et al. 1999]. The use of domain-dependent control knowledge has led to several successful planners, including TLPlan [Bacchus and Kabanza 2000], TALplan [Doherty and Kvarnstom 1999] and SHOP [Nau et al. 1999], which have performed very well on planning benchmarks. Strictly speaking, planners that use control knowledge are no longer considered to be classical planners since they require the addition of domain-dependent control knowledge to the problem specification. Nevertheless, such planners are predicted to be the most scalable types of planning systems in the long term [Wilkins and desJardines 2001]. In this paper we integrate the second and the third approaches identified above by translating a planning problem with domain-dependent control knowledge into a problem of model finding in logic programming.

We integrate domain-dependent control knowledge into our planner in such a way that planning can still be performed without this extra control knowledge. The control knowledge may simply improve the speed with which a plan is generated or may result in the generation of plans with particular desirable characteristics. In this respect[2] our approach is similar in spirit to the planning systems TLPlan [Bacchus and Kabanza 2000], and TALplan [Doherty and Kvarnstom 1999], but differs from typical Hierarchical Task Network (HTN) planners (e.g., SHOP [Nau et al. 1999]) because HTN planners require integration of domain-dependent control

[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.

[2]We differ from [Bacchus and Kabanza 2000] in another aspect. Unlike in [Bacchus and Kabanza 2000] where the domain knowledge is used by the search program thus controlling the search, our domain knowledge is encoded as a logic program which is directly added to the logic program encoding planning. In such an approach there is no guarantee that the added rules will reduce the search during answer set computation; although our experimentation shows that it does for large plan lengths. The paper [Huang et al. 1999] also comments on this aspect.

knowledge into the specification of the planning problem. As such, HTN planning cannot be performed without the existence of this knowledge, unlike our planner.

In this paper, we explore three kinds of domain control knowledge: temporal knowledge, procedural knowledge, and HTN-based knowledge. Our treatment of temporal knowledge is similar to that used in both the TLPlan and TALplan systems. Our formulation of procedural knowledge is inspired by GOLOG, referred to alternatively as a logic programming language, or an action execution language [Levesque et al. 1997]. Although our procedural knowledge is similar to the syntax of a GOLOG program, how this knowledge is used in planning is quite different. Similarly, our formulation of HTN-based knowledge is inspired by the partial-ordering constructs used in HTN planners, but our use of this type of knowledge during planning is very different from the workings of HTN planners. The main difference is that both GOLOG programming and HTN planning rely on the existence of domain-dependent control knowledge within the problem specification and cannot perform classical planning in the absence of this knowledge. In contrast, our approach, which is similar to the approach in [Bacchus and Kabanza 2000], separates the planner module from the domain knowledge (encoding temporal, procedural, or HTN-based knowledge), and can plan independent of the domain knowledge.

To achieve our goal of planning using domain-dependent control 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-dependent control knowledge. This leads to the question of choosing an appropriate language for both reasoning and representation tasks. For this we choose the action language $\mathcal{B}$ from [Gelfond and Lifschitz 1998] and the language of logic programming with answer set semantics (AnsProlog*) [Baral 2003], also referred to as A-Prolog [Gelfond and Leone 2002]. We discuss our choice on $\mathcal{B}$ in Section 2. We selected AnsProlog* over other action languages for a number of important reasons, many of which are listed below. These points are elaborated upon in [Baral 2003].

—AnsProlog* is a non-monotonic language that is suitable for knowledge representation. It is especially well-suited to reasoning in the presence of incomplete knowledge.

—The non-classical constructs give a structure to AnsProlog* programs and statements, such as a head and a body, which allows us to define various subclasses of the language, each with different complexity and expressivity properties [Dantsin et al. 2001]. The subclass of AnsProlog* programs in which no classical negation is allowed has the same complexity as propositional logic, but with added expressivity. The most general class of AnsProlog* programs, which allows "**or**" in the head, has the complexity and expressivity of the seemingly more complicated default logic [Reiter 1980]. In general, AnsProlog* is syntactically simpler than other non-monotonic logics while equally as expressive as many.

—There exists a sizable body of "building block" results about AnsProlog* which we may leverage both in knowledge representation tasks and in the analysis of the correctness of the representations. This includes result about composition of several AnsProlog* 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 AnsProlog* program to smaller components for computing its answer sets, and proving properties about the original program.

—There exist several efficient AnsProlog* interpreters [Eiter et al. 1998; Simons et al. 2002] and AnsProlog* 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, AnsProlog* has already been used in planning [Subrahmanian and Zaniolo 1995; Dimopoulos et al. 1997; Lifschitz 1999b], albeit in the absence of domain-dependent control knowledge. In this regard AnsProlog* is suitable for concisely expressing the effects of actions and static causal relations between fluents. Note that concise expression of the effects of actions requires addressing 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 AnsProlog* is a suitable candidate for model-based planning, and falls under category (ii) (above) 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-dependent control knowledge. In particular, our concerns 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 kinds of domain constraints, and

—correctness results for the task of planning using an AnsProlog* representation that includes domain constraints.

We also perform a limited number efficiency experiments, but leave more detailed experimentation to future work.

With the above focus, the contributions of the paper and the outline of the paper is as follows:

(1) In Section 3 we encode planning (without domain constraints) using AnsProlog* in the presence of both dynamic effects of actions and static causal laws, and with goals expressed as a conjunction of fluent literals. We then formally prove the relationship between valid trajectories of the action theory and answer sets of the encoded program. Our approach is similar to [Lifschitz and Turner 1999; Eiter et al. 2000] but differs from [Subrahmanian and Zaniolo 1995; Dimopoulos et al. 1997; Lifschitz 1999b]. The main difference between our formulation and earlier AnsProlog* encodings in [Subrahmanian and Zaniolo 1995; Dimopoulos et al. 1997; Lifschitz 1999b] is in our use of static causal laws, and our consideration of trajectories instead of plans. Our trajectories are similar to histories in [Lifschitz and Turner 1999] and to optimistic plans in [Eiter et al. 2000]. The reason we relate answer sets to trajectories rather

than relating them to plans is because in the presence of static causal laws the effects of actions may be non-deterministic.

(2) In Section 4.1 we show how to incorporate temporal constraints for planning into our formulation of the planning problem. Incorporating temporal constraints simply requires the addition of a few more rules, illustrating the declarative nature and elaboration tolerance of our approach. We define formulas for representing temporal constraints and specify when a trajectory satisfies a temporal constraint. We then formally prove the relationship between valid trajectories of the action theory satisfying the temporal constraints, and answer sets of the updated program. Our approach differs from [Bacchus and Kabanza 2000; Doherty and Kvarnstom 1999] in that we use AnsProlog* for both the basic encoding of planning and the temporal constraints, while the planners in [Bacchus and Kabanza 2000; Doherty and Kvarnstom 1999] are written in procedural languages. Preliminary experiments show that our approach is less efficient than TLPlan and TALPlan. Nevertheless, both these systems are highly optimized, so the poorer performance may simply reflect the lack of optimizations in our implementation. On the other hand, our use of AnsProlog* facilitates the provision of correctness proofs, which is one of our major concerns. Neither of [Bacchus and Kabanza 2000; Doherty and Kvarnstom 1999] provide correctness proofs of their planners.

(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 $\mathtt{a}_1\mathtt{;}\ \mathtt{a}_2\mathtt{;}\ \mathtt{(a}_3\ \mathtt{|}\ \mathtt{a}_4\ \mathtt{|}\ \mathtt{a}_5\mathtt{);f?}$. 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 for additional constructs from HTN planning to express domain knowledge and integrate features of HTNs 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 noted above, a significant contribution of our work is the suite of correctness proofs for our AnsProlog* formulations. All the proofs appear in Appendix A.

For completeness Appendix B presents results concerning AnsProlog* that we use in the Appendix A proofs.

In regards to closely related work, although satisfiability planning (see e.g., [Kautz and Selman 1992; Kautz et al. 1994; Kautz and Selman 1996; 1998a]) 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 AnsProlog*, which will be used in the rest of the paper.

2. PRELIMINARIES AND BACKGROUND

In this section, we review the basics of the action description language $\mathcal{B}$, the answer set semantics of logic programs (AnsProlog), and key features of problem solving using AnsProlog.

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 first be 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 action 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. In this paper, we adopt the language $\mathcal{B}$ [Gelfond and Lifschitz 1998], which is a subset of the language proposed in [Turner 1997], for its simple syntax and its capability to represent relationships between fluents, an important feature lacking in many variants of the action description language $\mathcal{A}$ [Gelfond and Lifschitz 1993]. We note that the main results of this paper can be used in answer set planning systems which use other languages for representing and reasoning about the effects of actions.

We now present the basics of the action description language $\mathcal{B}$. An action theory in $\mathcal{B}$ is defined over two disjoint sets, a set of actions $\mathbf{A}$ and a set of fluents $\mathbf{F}$, which are defined over a signature $\sigma = \langle \mathbf{O}, \mathbf{AN}, \mathbf{FN} \rangle$ where

—$\mathbf{O}$ is a finite set of object constants;
—$\mathbf{AN}$ is a finite set of *action names*, each action name is associated with a number n, $n \geq 0$, which denotes its arity; and
—$\mathbf{FN}$ is a finite set of *fluent names*, each fluent name is associated with a number n, $n \geq 0$, which denotes its arity.

An action in $\mathbf{A}$ is of the form $a(c_1, \ldots, c_n)$ where $a \in \mathbf{AN}$ is a n-ary action name and c_i is a constant in $\mathbf{O}$. A fluent in $\mathbf{F}$ is of the form $f(c_1, \ldots, c_m)$ where $f \in \mathbf{FN}$ is an m-ary fluent name and c_i is a constant in $\mathbf{O}$. For simplicity, we often write a and f to represent an action or a fluent whenever it is unambiguous. Furthermore, we will omit the specification of σ when it is clear from the context.

A *fluent literal* is either a fluent $f \in \mathbf{F}$ or its negation $\neg f$. A *fluent formula* is a propositional formula constructed from fluent literals. An action theory is 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 and a is an action. (1) represents a *static causal law*, i.e., a relationship between fluents[3]. It conveys that whenever the fluent literals $p_1, \ldots, p_n$ hold in a state, that causes f to also hold in that state. (2), referred to as a *dynamic causal law*, represents the (conditional) effect of a while (3) encodes an executability condition of a. Intuitively, an executability condition of the form (3) states that a can only be executed if p_i's holds. A dynamic law of the form (2) states that f is caused to be true after the execution of a in any state of the world where $p_1, \ldots, p_n$ are true. Propositions of the form (4) are used to describe the initial state. They state that f holds in the initial state.

An *action theory* is a pair (D, Γ) where Γ, called the *initial state*, 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 2.1. Let us consider a modified version of the suitcase s with two latches from [Lin 1995]. We have a suitcase with two latches l_1 and l_2. l_1 is up and l_2 is down. To open a latch (l_1 or l_2) we need a corresponding key (k_1 or k_2, respectively). When the two latches are in the up position, the suitcase is unlocked. When one of the latches is down, the suitcase is locked. The signature of this domain consists of

—$\mathbf{O} = \{l_1, l_2, s, k_1, k_2\}$;
—$\mathbf{AN} = \{open, close\}$, both action names are associated with the number 1, and
—$\mathbf{FN} = \{up, locked, holding\}$, all fluent names are associated with the number 1.

In this domain, we have that

$$\mathbf{A} = \{open(l_1), open(l_2), close(l_1), close(l_2)\}$$

and

$$\mathbf{F} = \{locked(s), up(l_2), up(l_1), holding(k_1), holding(k_2)\}.$$

We now present the propositions describing the domain.

[3]A constraint between fluents can also be represented using static causal laws. For example, to represent the fact that a door cannot be *opened* and *closed* at the same time, i.e. the fluents *opened* and *closed* cannot be true at the same time, we introduce a new fluent, say *inconsistent*, and represent the constraint by two static causal laws $\mathbf{caused}(\{opened, closed\}, inconsistent)$ and $\mathbf{caused}(\{opened, closed\}, \neg inconsistent)$.

Opening a latch puts it into the up position. This is represented by the dynamic laws:

$$\textbf{causes}(open(l_1), up(l_1), \{\}) \text{ and } \textbf{causes}(open(l_2), up(l_2), \{\}).$$

Closing a latch puts it into the down position. This can be written as:

$$\textbf{causes}(close(l_1), \neg up(l_1), \{\}) \text{ and } \textbf{causes}(close(l_2), \neg up(l_2), \{\}).$$

We can open the latch only when we have the key. This is expressed by:

$$\textbf{executable}(open(l_1), \{holding(k_1)\}) \text{ and } \textbf{executable}(open(l_2), \{holding(k_2)\}).$$

No condition is required for closing a latch. This is expressed by the two propositions:

$$\textbf{executable}(close(l_1), \{\}) \text{ and } \textbf{executable}(close(l_2), \{\}).$$

The fact that the suitcase will be unlocked when the two latches are in the up position is represented by the static causal law:

$$\textbf{caused}(\{up(l_1), up(l_2)\}, \neg locked(s)).$$

Finally, to represent the fact that the suitcase will be locked when either of the two latches is in the down position, we use the following static laws:

$$\textbf{caused}(\{\neg up(l_1)\}, locked(s)) \text{ and } \textbf{caused}(\{\neg up(l_2)\}, locked(s))$$

The initial state of this domain is given by

$$\Gamma = \left\{ \begin{array}{l} \textbf{initially}(up(l_1)) \\ \textbf{initially}(\neg up(l_2)) \\ \textbf{initially}(locked(s)) \\ \textbf{initially}(\neg holding(k_1)) \\ \textbf{initially}(holding(k_2)) \end{array} \right\}$$

□

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 the execution of a in s results in an error. We now formally define Φ.

Let D be a domain description in $\mathcal{B}$. A set of fluent literals is said to be *consistent* if it does not contain f and $\neg f$ for some fluent f. An *interpretation* I of the fluents in D is a maximal consistent set of fluent literals of D. 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.

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 law

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

in K, if $u \models p_1 \wedge \ldots \wedge p_n$ then $u \models f$. By $Cl_K(u)$ we denote the least consistent set of literals from D that contains u and is also closed under K. It is worth noting that $Cl_K(u)$ might be undefined. For instance, if u contains both f and $\neg f$ for some fluent f, then $Cl_K(u)$ cannot contain u and be consistent; another example is that if $u = \{f, g\}$ and K contains

$$\textbf{caused}(\{f\}, h)$$

and

$$\textbf{caused}(\{f, g\}, \neg h),$$

then $Cl_K(u)$ does not exist because it has to contain both h and $\neg h$, which means that it is inconsistent.

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

$$\textbf{executable}(a, \{f_1, \ldots, f_n\})$$

in D such that $s \models f_1 \wedge \ldots \wedge f_n$. Clearly, if

$$\textbf{executable}(a, \{\})$$

belongs to D, then a is executable in every state of D.

The *direct effect of an action* a in a state s is the set

$$E(a, s) = \{f \mid \textbf{causes}(a, f, \{f_1, \ldots, f_n\}) \in D, s \models f_1 \wedge \ldots \wedge f_n\}.$$

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 they were not changed by an 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 [McCain and Turner 1995]. Additional explanations and motivations behind the above definition can be found in [Baral 1995; McCain and Turner 1995; Turner 1997].

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

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 2.2. For the suitcase domain in Example 2.1, the initial state given by the set of propositions

$$\Gamma = \begin{cases} \textbf{initially}(up(l_1)) \\ \textbf{initially}(\neg up(l_2)) \\ \textbf{initially}(locked(s)) \\ \textbf{initially}(\neg holding(k_1)) \\ \textbf{initially}(holding(k_2)) \end{cases}$$

is

$$s_0 = \{up(l_1), \neg up(l_2), locked(s), \neg holding(k_1), holding(k_2)\}.$$

In state s_0, the three actions $open(l_2)$, $close(l_1)$, and $close(l_2)$ are executable. $open(l_2)$ is executable since $holding(k_2)$ is true in s_0 while $close(l_1)$ and $close(l_2)$ are executable since the theory (implicitly) contains the propositions:

$$\textbf{executable}(close(l_1), \{\}) \qquad \text{and} \qquad \textbf{executable}(close(l_2), \{\})$$

which indicate that these two actions are always executable. The following transitions are possible from state s_0:

$$\{ up(l_1), up(l_2), \neg locked(s), \neg holding(k_1), holding(k_2) \} \in \Phi(open(l_2), s_0).$$
$$\{ up(l_1), \neg up(l_2), locked(s), \neg holding(k_1), holding(k_2) \} \in \Phi(close(l_2), s_0).$$
$$\{ \neg up(l_1), \neg up(l_2), locked(s), \neg holding(k_1), holding(k_2) \} \in \Phi(close(l_1), s_0).$$

□

For a domain description D with transition function Φ, a sequence $s_0 a_0 s_1 \dots a_{n-1} s_n$ 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 $i \in \{0, \dots, n-1\}$. A trajectory $s_0 a_0 s_1 \dots a_{n-1} s_n$ 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[5] consistent action theories.

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 $a_0, \dots, a_{m-1}$ is then called a *possible plan for* Δ if there exists a trajectory $s_0 a_0 s_1 \dots a_{m-1} s_m$ 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'. This is because the presence of static causal laws in D allows the possibility that the

[5]We thank one of the anonymous referee for pointing out that without this assumption, finding a plan would be $\Sigma_3\mathbf{P}$-complete even with respect to a complete initial state.

effects of actions may be non-deterministic, and planning with non-deterministic actions has the complexity of $\Sigma_2\mathbf{P}$-complete [Turner 2002] and hence is beyond the expressiveness of AnsProlog. However, if D is deterministic, i.e., $|\Phi(a, s)| \leq 1$ for every pair (a, s) of actions and states, then the notions of 'possible plan' and 'plan' coincide.

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

In this section we review AnsProlog (a sub-class of AnsProlog*) and its applicability to problem solving.

2.2.1 *AnsProlog.* 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 only proposed by Gelfond and Lifschitz in 1988 [Gelfond and Lifschitz 1988]. 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. Initially, some considered the existence of multiple or no stable models to be a drawback of stable model semantics, while others considered it to be a reflection of the poor quality of the program in question. Nevertheless, it is this feature of the answer set semantics [Marek and Truszczyński 1999; Niemelä 1999; Lifschitz 1999b] that is key to the use of AnsProlog for problem solving. We now present the syntax and semantics of AnsProlog, which we will simply refer to as a logic program.

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* is a connective called *negation-as-failure*. 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 of the form (5) whose body is satisfied by X.

For a set of ground atoms S and a program Π, the reduct of Π with respect to S, denoted by Π^S, 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 that satisfies all the rules in Π.

(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** [Eiter et al. 1998] and **smodels** [Simons et al. 2002]. Recently, two new systems **cmodels** [Babovich and Lifschitz] and **ASSAT** [Lin and Zhao 2002], which compute answer sets by using SAT solvers, have been developed. **XSB** [Sagonas et al. 1994], 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 *Answer set programming: problem solving using AnsProlog.* Prolog and other early logic programming systems were geared towards answering yes/no queries with respect to a program, and if the queries had variables they returned instantiations of the variables together with a 'yes' answer. The possibility of multiple answer sets and no answer sets has given rise to an alternative way to solve problems using AnsProlog. In this approach, referred to as answer set programming (also known as stable model programming) [Marek and Truszczyński 1999; Niemelä 1999; Lifschitz 1999b], possible solutions of a problem are enumerated as answer set candidates and non-solutions are eliminated through rules with $\bot$ 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 AnsProlog. 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:

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

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. [Niemelä et al. 1999] 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 [Niemelä et al. 1999] where a program with weight constraint rules is translated into a normal logic program whose answer sets define the answer sets of the original program. For our purpose in this paper we only need to consider rules with $l \leq 1$, $u = 1$, 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.

3. ANSWER SET PLANNING: USING ANSPROLOG FOR PLANNING

The idea of using logic programming with answer set semantics for planning was first introduced in [Subrahmanian and Zaniolo 1995]. It has become more feasible since the development of fast and efficient answer set solvers such as **smodels** [Simons et al. 2002] and **dlv** [Eiter et al. 1998]. The term "*answer set planning*" was coined by Lifschitz in [Lifschitz 1999b] referring to approaches to planning using logic programming with answer set semantics, where the planning problem is expressed as a logic program and the answer sets encode plans. In that paper answer set planning is illustrated with respect to some specific examples.

We now present the main ideas of answer set planning[6] when the effects of actions on the world and the relationships between fluents in the world are expressed in

[6]Note that while [Lifschitz 1999b; 1999a; 2002] illustrated answer set planning through specific examples, the papers [Lifschitz and Turner 1999; Lifschitz 1999a] mapped reasoning (not planning) in the action description language $\mathcal{C}$ to logic programming.

the action description language $\mathcal{B}$. 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 inertial law. We assume that actions and fluents in $\mathbf{A}$ and $\mathbf{F}$ are specified by facts of the form $action(.)$ and $fluent(.)$, respectively. The rules of Π are adapted mainly from [Dimopoulos et al. 1997; Lifschitz 1999b; Lifschitz and Turner 1999; Lifschitz 1999a] and based on conversations with Michael Gelfond in 1998-99. As customary in the encoding of planning problems, we assume that the length of plans we are looking for is given. We denote the length by the constant *length* and use a sort *time*, whose domain is the set of integers from 0 to *length*, to represent the time moment the system is in. We begin with the set of domain-dependent rules.

3.1 Domain-dependent rules

For an action theory (D, Γ), the encoding of (D, Γ) uses the following predicates:

(1) $holds(F, T)$ – the fluent literal F holds at the time moment T;
(2) $occ(A, T)$ – the action A occurs at the time moment T; and
(3) $possible(A, T)$ – the action A is executable at the time moment T.

The translation is as follows[7].

(1) For each proposition

$$\mathbf{initially}(l)$$

in Γ, the fact

$$holds(l, 0) \tag{8}$$

belongs to Π. This says that at the time moment 0, the fluent literal l holds.

(2) For each executability condition

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

in D, Π contains the rule:

$$possible(a, T) \leftarrow time(T), holds(p_1, T), \ldots, holds(p_n, T). \tag{9}$$

This rules states that it is possible to execute the action a at the time moment T if $\{p_1, \ldots, p_n\}$ holds at T.

(3) For each dynamic causal law

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

in D, Π contains the rule:

$$\begin{aligned} holds(f, T+1) \leftarrow\ & time(T), occ(a, T), \\ & possible(a, T), holds(p_1, T), \ldots, holds(p_n, T). \end{aligned} \tag{10}$$

[7] A **Sicstus**-program that translates $\mathcal{B}$ planning problems into their corresponding **smodels** encodings is available at `http://www.cs.nmsu.edu/~tson/ASPlan/Knowledge`. (An earlier version of this translator was posted to the TAG discussion web site `http://www.cs.utexas.edu/users/vl/tag/discussions.html`).

This rule says that if a occurs at the time moment T and a is executable at T then the fluent literal f becomes true at $T+1$ if there exists a dynamic law

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

in D and the p_i's hold at T.

(4) For each static causal law

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

in D, Π contains the following rule:

$$holds(f, T) \leftarrow time(T), holds(p_1, T), \ldots, holds(p_n, T). \tag{11}$$

This rule is a straightforward translation of the static causal law into logic programming rule.

We demonstrate the above translation by encoding the blocks world domain from Example 2.1.

EXAMPLE 3.1. The rules encoding the fluents and actions of the suitcase domain in Example 2.1 are:

$$\begin{array}{ll} action(open(l_1)) \leftarrow & fluent(up(l_1)) \leftarrow \\ action(open(l_2)) \leftarrow & fluent(up(l_2)) \leftarrow \\ action(close(l_1)) \leftarrow & fluent(locked(s)) \leftarrow \\ action(close(l_2)) \leftarrow & fluent(holding(k_1))) \leftarrow \\ & fluent(holding(k_2))) \leftarrow \end{array}$$

The first group of rules (left column) define the set $\mathbf{A}$ and the second group of rules (right column) define $\mathbf{F}$. The dynamic law $\mathbf{causes}(open(l_1), up(l_1), \{\})$ is translated into the rule:

$$holds(up(l_1), T+1) \leftarrow time(T), occ(open(l_1), T).$$

The dynamic law $\mathbf{causes}(close(l_1), \neg up(l_1), \{\})$ is translated into the rule:

$$holds(\neg up(l_1), T+1) \leftarrow time(T), occ(close(l_1), T).$$

The executability condition $\mathbf{executable}(open(l_1), \{holding(k_1)\})$ is translated into the rule:

$$possible(open(l_1), T) \leftarrow time(T), holds(holding(k_1), T).$$

The static causal law $\mathbf{caused}(\{up(l_1), up(l_2)\}, \neg locked(s))$ is encoded by the rule:

$$holds(\neg locked(s), T) \leftarrow time(T), holds(up(l_1), T), holds(up(l_2), T).$$

The static causal law $\mathbf{caused}(\{\neg up(l_1)\}, locked(s))$ is encoded by the rule:

$$holds(locked(s), T) \leftarrow time(T), holds(\neg up(l_1), T).$$

The encoding of other propositions of the domain is similar. □

3.2 Domain independent rules

The set of domain independent rules of Π consists of rules for generating action occurrences and rules for defining auxiliary predicates. First, we present the rules for the generation of action occurrences.

$$occ(A,T) \leftarrow action(A), time(T), possible(A,T), not\ nocc(A,T). \quad (12)$$
$$nocc(A,T) \leftarrow action(A), action(B), time(T), A \neq B, occ(B,T). \quad (13)$$

In the above rules, A and B are variables representing actions. These rules generate action occurrences, one at a time [8]. The rules of inertia (or the frame axioms) and rules defining literals are encoded using the following rules:

$$literal(F) \leftarrow fluent(F). \quad (14)$$
$$literal(\neg F) \leftarrow fluent(F). \quad (15)$$
$$contrary(F, \neg F) \leftarrow fluent(F). \quad (16)$$
$$contrary(\neg F, F) \leftarrow fluent(F). \quad (17)$$
$$holds(L, T{+}1) \leftarrow literal(L), literal(G), time(T), \quad (18)$$
$$contrary(L,G), holds(L,T), not\ holds(G, T{+}1).$$

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 cannot be true at the same time, the following constraint is added to Π.

$$\bot \leftarrow fluent(F), holds(F,T), holds(\neg F, T). \quad (19)$$

3.3 Goal representation

The goal Δ is encoded by two sets of rules. The first set of rules defines Δ as a formula over fluent literals and the second set of rules evaluates the truth value of Δ at different time moments. In a later section, we show how fluent formulas can be represented and evaluated. In this section, we will assume that Δ is simply a conjunction of literals, i.e.,

$$\Delta = p_1 \wedge \ldots \wedge p_k$$

where p_i are literals. Then, Δ is represented by the following rules:

$$goal \leftarrow holds(p_1, n), \ldots, holds(p_k, n). \quad (20)$$

(Recall that the constant n denotes the maximal length of trajectories that we are looking for.)

[8] 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 constraints that requires that actions can occur only when they are executable and when some actions are executable then one must occur. In many of our experiments, programs with these rules yield better performance.

3.4 Correctness of Π

Let $\Pi_n(D, \Gamma, \Delta)$ be the logic program consisting of

—the set of rules encoding D and Γ (rules (8)-(11)) in which the domain of T is $\{0, \ldots, n\}$ and the rules define actions and fluents of (D, Γ),
—the set of domain-independent rules (rules (12)-(19)) in which the domain of T is $\{0, \ldots, n\}$,
—the rule in (20) and the constraint $\bot \leftarrow not\ goal(n)$ that encodes the requirement that Δ holds at n.

In what follows, we will write Π_n instead of $\Pi_n(D, \Gamma, \Delta)$ when it is clear from the context what D, Γ, and Δ are. The following result (similar to the correspondence between histories and answer sets in [Lifschitz and Turner 1999]) 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 3.2. 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 $s_0(M) a_0 \ldots a_{k-1} s_k(M)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$ and if $k < n$ then no action is executable in the state $s_k(M)$.

PROOF. See Appendix A.1 □

Note that the second item of the theorem implies that the trajectory achieving Δ corresponds to an answer set M of Π_n that 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.

Recall that the sequence of actions $a_0, a_1, \ldots, a_{n-1}$, where $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trajectory achieving Δ, is *not necessarily a plan achieving the goal* Δ because the action theory (D, Γ) may be non-deterministic. It is easy to see that whenever (D, Γ) is deterministic, if $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trajectory achieving Δ then $a_0, a_1, \ldots, a_{n-1}$ is indeed a plan achieving Δ. The next corollary follows directly from Theorem 3.2.

COROLLARY 3.3. For a planning problem $\langle D, \Gamma, \Delta\rangle$ with a consistent and deterministic action theory (D, Γ),

(1) for each plan $a_0, \ldots, a_{n-1}$ achieving Δ from Γ, 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 $0 \leq i < k$ and if $k < n$ then no action is executable in the state reached after executing $a_0, \ldots, a_{k-1}$ in the initial state.

The first item of the corollary follows from Item (i) of Theorem 3.2. Nevertheless, we do not need to include the condition on the states $s_i(M)$ because $s_i(M)$ is uniquely determined by the initial state s_0 and the sequence of actions $a_0, \ldots, a_{i-1}$. The second item of the corollary follows from Item (ii) of Theorem 3.2. Again, because of the determinism of (D, Γ), we do not need to include the conditions on the states $s_i(M)$.

4. CONTROL KNOWLEDGE AS CONSTRAINTS

In this section, we present the main contribution of this paper: augmenting the answer set planning program Π, introduced in the previous section, with different kinds of domain knowledge, namely temporal knowledge, procedural knowledge, and HTN-based knowledge. The domain knowledge acts 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 now proceed to introduce the different types of control knowledge. We start with temporal knowledge.

4.1 Temporal Knowledge

Use of temporal domain knowledge in planning was first proposed by Bacchus and Kabanza in [Bacchus and Kabanza 2000]. 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 the search process have proven to be highly efficient and to scale up well [Bacchus et al. 2000]. In this paper, we represent temporal knowledge using temporal constraints. Temporal constraints are built from fluent formulae using the *temporal operators* **always**, **until**, **next**, and **eventually**, and a special *goal operator* **goal**. For simplicity of the presentation, we will write fluent formulae in prefix notation and use the *propositional connectives* **and, or**, and **negation**. Given a signature $\langle \mathbf{O}, \mathbf{AN}, \mathbf{BN} \rangle$ we define term, formula, and closed formula, as follows.

Definition 4.1. A *term* is a variable or a constant in $\mathbf{O}$.

Definition 4.2. A formula is either

—an expression of the form $f(\sigma_1, \ldots, \sigma_n)$ where f is a n-ary fluent name and each σ_i is a term;

—an expression of the form $\mathbf{and}(\phi, \psi)$, where ϕ and ψ are formulae;

—an expression of the form $\mathbf{or}(\phi, \psi)$, where ϕ and ψ are formulae;

—an expression of the form $\mathbf{negation}(\phi)$, where ϕ is a formula; or

—an expression of the form $(\exists X.\{c_1, \ldots, c_n\}\phi)$ or $(\forall X.\{c_1, \ldots, c_n\}\phi)$ where X is a variable, $\{c_1, \ldots, c_n\}$ is a set of constants in $\mathbf{O}$, and ϕ is a formula.

We next define the notion of a *closed formula*.

Definition 4.3. The formula over which a quantifier applies is called the *scope* of the quantifier. The scope of $\forall X.\{c_1, \ldots, c_n\}$ (resp. $\exists X.\{c_1, \ldots, c_n\}$) in

$(\forall X.\{c_1,\ldots,c_n\}\phi)$ (resp. $(\exists X.\{c_1,\ldots,c_n\}\phi)$) is ϕ. An occurrence of a variable in a formula is a bounded occurrence iff the occurrence is within the scope of a quantifier which has the same variable immediately after the quantifier or is the occurrence of that quantifier. An occurrence of a variable in a formula is a free occurrence iff the occurrence is not bound. A variable is free in a formula if at least one of its occurrences is a free occurrence.

Definition 4.4. A formula without free variables is called a *closed formula*.

Remark 4.5. The truth or falsity of a formula is evaluated with respect to state in the standard way. It is easy to see that formulae with quantifiers can be translated into equivalent formulae without quantifier as follows: $\forall X.\{c_1,\ldots,c_n\}\phi$ is equivalent to $\bigwedge_{i=1}^{n}\phi(c_i)$ and $\exists X.\{c_1,\ldots,c_n\}\phi$ is equivalent to $\bigvee_{i=1}^{n}\phi(c_i)$ where $\phi(c_i)$ is the formula obtained from ϕ by replacing every free occurrence of X in ϕ with c_i. For this reason, we will be dealing with formulae without quantifiers hereafter.

We are now ready to define the notion of a temporal constraint.

Definition 4.6. A *temporal constraint* is either

—a closed formula (Definition 4.4)

—an expression of the form **goal**(ϕ) where ϕ is a closed formula; or

—an expression of the form **and**(ϕ,ψ), **or**(ϕ,ψ), **negation**(ϕ), **until**(ϕ,ψ), **always**(ϕ), **eventually**(ϕ), or **next**(ϕ) where ϕ and ψ are temporal constraints.

A temporal constraint is an *atomic* constraint if it is a fluent literal. Otherwise, it is called *non-atomic*. In what follows, a constraint ϕ will be referred as a *sub-constraint* of a constraint ψ if ϕ occurs in ψ. We will write $sub(\phi)$ to denote the set of constraints consisting of ϕ and its sub-formulae. It is easy to see that constraints without temporal operators or the goal operator are indeed *fluent formulae*. Temporal operators are understood with their standard meaning while the goal operator **goal** provides a convenient way for expressing the control knowledge which depends on goal information. A temporal constraint is said to be goal-independent if no goal formula occurs in it. Otherwise, it is goal-dependent. Bacchus and Kabanza [Bacchus and Kabanza 2000] observed that useful temporal knowledge in planning is often goal-dependent. In the blocks world domain, the following goal-dependent constraint[9]:

$$\textbf{always}(\textbf{and}(\textbf{goal}(on(X,tbl)),on(X,tbl))\supset\textbf{next}(on(X,tbl))) \qquad (21)$$

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 on the table in the goal state.

Notice 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, **always**(**next**(φ)) is a temporal formula, but **goal**(**goal**(φ)) is not.

[9]Because material implication (denoted by $\supset$) can be replaced by $\vee$ and $\neg$, we omit it in the definition but use it in writing the constraints, to simplify reading. As before, we use the convention that a formula with variables represents the set of its ground instantiations.

Goal-independent formulae will be interpreted over an infinite sequence of states of D, denoted by $I = \langle s_0, s_1, \ldots, \rangle$. On the other hand, goal-dependent formulae will be evaluated with respect to a pair $\langle I, \varphi \rangle$ where I is a sequence of states and φ is a fluent formula. In the next two definitions, we formally define when a constraint is satisfied. Definition 4.7 deals with goal-independent constraints while Definition 4.8 is concerned with general constraints.

Definition 4.7. (See [Bacchus and Kabanza 2000]) 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 constraints, t be a non-negative integer, and f_3 be a fluent formula. Let $I_t = \langle s_t, s_{t+1}, \ldots, \rangle$ denote the subsequence of I starting from s_t. We say that I satisfies f (f is either f_1, f_2, or f_3), denoted by $I \models f$, iff $I_0 \models f$ where

—$I_t \models f_3$ iff $s_t \models f_3$.
—$I_t \models \mathbf{until}(f_1, f_2)$ iff there exists $t \leq t_2$ such that $I_{t_2} \models f_2$ and for all $t \leq t_1 < t_2$ 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 $t \leq t_1$ such that $I_{t_1} \models f_1$.
—$I_t \models \mathbf{always}(f_1)$ iff for all $t \leq t_1$ 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 constraint f, we say that I 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 goal-dependent temporal constraints are satisfied by a sequence of states and a goal. Intuitively, this should be a straightforward extension of the previous definition in which formulas of the form $\mathbf{goal}(\varphi)$ need to be accounted for. Obviously, such a constraint can only be evaluated with respect to a sequence of states and a formula encoding the goal. Furthermore, the intuition behind the formula $\mathbf{goal}(\psi)$ is that ψ is true whenever the goal is true, i.e., ψ is entailed by the goal. This is detailed in the second item of the following definition.

Definition 4.8. Let $I = \langle s_0, s_1, \ldots, s_n, \ldots \rangle$ be a sequence of states of D and φ be a fluent formula denoting the goal. Let f_1 and f_2 be temporal constraints (possibly goal dependent), t be a non-negative integer, and f_3 be a fluent formula. Let $I_t = \langle s_t, s_{t+1}, \ldots, \rangle$. We say that I satisfies f (f is either f_1, f_2, or f_3) with respect to φ, denoted by $\langle I, \varphi \rangle \models f$, iff $\langle I_0, \varphi \rangle \models f$ where

—$\langle I_t, \varphi \rangle \models f_3$ iff $s_t \models f_3$.
—$\langle I_t, \varphi \rangle \models \mathbf{goal}(f_3)$ iff $\varphi \models f_3$ [10].
—$\langle I_t, \varphi \rangle \models \mathbf{until}(f_1, f_2)$ iff there exists $t \leq t_2$ such that $\langle I_{t_2}, \varphi \rangle \models f_2$ and for all $t \leq t_1 < t_2$ 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 $t \leq t_1$ such that $\langle I_{t_1}, \varphi \rangle \models f_1$.
—$\langle I_t, \varphi \rangle \models \mathbf{always}(f_1)$ iff for all $t \leq t_1$ we have $\langle I_{t_1}, \varphi \rangle \models f_1$.

[10]Here, by $\varphi \models f_3$ we mean that φ entails f_3.

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

To complete the encoding of temporal constraints, we now provide the rules that check the satisfiability of a temporal constraint given a trajectory. We define the predicate $hf(F, T)$ whose truth value determines whether F 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. It is easy to see that rules for checking the satisfiability of temporal constraints can be straightforwardly developed in logic programming with function symbols. For example, the rules

$$\begin{aligned} hf(L, T) &\leftarrow holds(L, T), literal(L) \\ hf(and(F_1, F_2), T) &\leftarrow hf(F_1, T), hf(F_2, T) \end{aligned}$$

can be used to determine whether or not the constraint $\mathbf{and}(F_1, F_2)$ is true at the time moment T. The first rule is for atomic constraints and the second rule is for non-atomic ones. Although these rules are intuitive and correct, we will need to modify them for use with the currently available answer set solvers such as **dlv** and **smodels**. This is because **dlv** does not allow function symbols and **lparse** – the parser of the **smodels** system – requires that variables occurring in the head of a rule are domain variables, i.e., in the second rule, we have to specify the domain of F_1 and F_2.

We will now present two possible ways to deal with the answer set solver's restriction[11]. The first way is to represent a constraint by a set of rules that determine its truth value. In other words, we specify the domains of F_1 and F_2 in the above rules by grounding them. For example, for the conjunction $\mathbf{and}(f, g)$, the rules

$$\begin{aligned} hf(L, T) &\leftarrow literal(L), holds(L, T) \\ hf(and(f, g), T) &\leftarrow hf(f, T), hf(g, T) \end{aligned}$$

can be used. For the disjunction, $\mathbf{or}(f, \mathbf{and}(g, h))$, the rules

$$\begin{aligned} hf(L, T) &\leftarrow literal(L), holds(L, T) \\ hf(or(f, and(g, h)), T) &\leftarrow hf(f, T). \\ hf(or(f, and(g, h)), T) &\leftarrow hf(and(g, h), T). \\ hf(and(g, h), T) &\leftarrow hf(g, T), hf(h, T). \end{aligned}$$

can be used. The encodings of other constraints are similar. Observe that the number of rules for encoding a formula depends on the number of its sub-constraints.

An alternative to the above encoding is to assign names to non-atomic constraints, to define a new type, called *formula*, and to provide the constraint-independent

[11] Another alternative for dealing with temporal constraints such as fluent formulae is to convert them into disjunctive normal form and to develop, for each conjunction $\mathbf{and}(f_1, \mathbf{and}(f_2, \ldots, \mathbf{and}(f_{n-1}, f_n)))$, a rule

$$hf(\mathbf{and}(f_1, \mathbf{and}(f_2, \ldots, \mathbf{and}(f_{n-1}, f_n))), T) \leftarrow holds(f_1, T), \ldots, holds(f_n, T).$$

This method, however, cannot be easily extended for temporal constraints with temporal operators or the goal operator.

rules for checking the truth value of constraints. Atomic constraints are defined by the rule:

$$formula(L) \leftarrow literal(L).$$

For each non-atomic formula ϕ, we associate with it a unique name n_ϕ and encode it by a set of facts, denoted by $r(\phi)$. This set is defined inductively over the structure of ϕ as follows.

—If ϕ is a fluent literal l then $r(\phi) = \{l\}$;
—If $\phi = \phi_1 \wedge \phi_2$ then $r(\phi) = r(\phi_1) \cup r(\phi_2) \cup \{formula(n_\phi), and(n_\phi, n_{\phi_1}, n_{\phi_2})\}$;
—If $\phi = \phi_1 \vee \phi_2$ then $r(\phi) = r(\phi_1) \cup r(\phi_2) \cup \{formula(n_\phi), or(n_\phi, n_{\phi_1}, n_{\phi_2})\}$;
—If $\phi = \neg\phi_1$ then $r(\phi) = r(\phi_1) \cup \{formula(n_\phi), negation(n_\phi, n_{\phi_1})\}$;
—If $\phi = \mathbf{next}(\phi_1)$ then $r(\phi) = r(\phi) \cup \{formula(n_\phi), next(n_\phi, n_{\phi_1})\}$;
—If $\phi = \mathbf{until}(\phi_1, \phi_2)$ then

$$r(\phi) = r(\phi_1) \cup r(\phi_2) \cup \{formula(n_\phi), until(n_\phi, n_{\phi_1}, n_{\phi_2})\};$$

—If $\phi = \mathbf{always}(\phi_1)$ then $r(\phi) = r(\phi_1) \cup \{formula(n_\phi), always(n_\phi, n_{\phi_1})\}$;
—If $\phi = \mathbf{eventually}(\phi_1)$ then $r(\phi) = r(\phi_1) \cup \{formula(n_\phi), eventually(n_\phi, n_{\phi_1})\}$.

For simplicity, the names assigned to a constraint can be used in encoding other constraints. For example, the constraints $\phi = \mathbf{and}(f, \mathbf{and}(g, h))$ is encoded by the atoms

$$formula(n_\psi).$$
$$and(n_\psi, g, h).$$
$$formula(n_\phi).$$
$$and(n_\phi, f, n_\psi).$$

We note that the above encodings can be generated automatically using a program front-end to **smodels** that is available on the web-site containing the experimental results presented in this paper. Note that during the grounding phase of **smodels** (by **lparse**), atoms of the form $formula(.,.)$ will be removed. For this reason, we use the second encoding in our experiments because it is easier to deal with changes in the constraints used for encoding the control knowledge.

We now present the formula-independent rules for evaluating temporal constraints. As with defining the satisfaction of temporal constraints, we first consider goal-independent temporal constraints. The rules needed for evaluating temporal constraints whose first level operator is different than the **goal** operator are as follows:

$$hf(L,T) \leftarrow literal(L), holds(L,T). \quad (22)$$
$$hf(N,T) \leftarrow formula(N), and(N, N_1, N_2), \quad (23)$$
$$hf(N_1,T), hf(N_2,T). \quad (24)$$
$$hf(N,T) \leftarrow formula(N), or(N, N_1, N_2), hf(N_1,T). \quad (25)$$
$$hf(N,T) \leftarrow formula(N), or(N, N_1, N_2), hf(N_2,T). \quad (26)$$

$$hf(N,T) \leftarrow formula(N), negation(N,N_1)), not\ hf(N_1,T). \quad (27)$$
$$hf(N,T) \leftarrow formula(N), until(N,N_1,N_2), \quad (28)$$
$$T \leq T', hf\_during(N_1,T,T'), hf(N_2,T').$$
$$hf(N,T) \leftarrow formula(N), always(N,N_1), \quad (29)$$
$$hf\_during(N_1,T,n).$$
$$hf(N,T) \leftarrow formula(N), eventually(N,N_1), \quad (30)$$
$$hf(N_1,T'), T \leq T'.$$
$$hf(N,T) \leftarrow formula(N), next(N,N_1), hf(N_1,T+1). \quad (31)$$
$$hf\_during(N,T,T) \leftarrow hf(N,T). \quad (32)$$
$$hf\_during(N,T,T') \leftarrow hf(N,T), T < T', hf\_during(N,T+1,T'). \quad (33)$$

The meaning of these rules is straightforward. The first rule defines the truth value of an atomic formula (a literal). Rule (23) says that a conjunction holds if its conjuncts hold. Rules (25)-(26) say that a disjunction holds if one of its disjuncts holds. The rule (27) states that the negation of a formula holds if its negation does not hold. Its correctness is due to the assumption that initial states are complete. Rules (28)-(33) deal with formulae containing temporal operators. The constant n denotes the maximal length of trajectories that we are looking for. In the following, we refer to this group of rules by $\Pi_{formula}$.

The next theorem shows that rules (22)-(33) correctly implement the semantics of goal-independent temporal formulae.

THEOREM 4.9. Let S be a finite set of goal-independent temporal formulae, $I = \langle s_0, s_1 \dots s_n \rangle$ be a sequence of states, and

$$\Pi_{formula}(S,I) = \Pi_{formula} \cup r(I) \cup r(S)$$

where

—$r(S)$ is the set of atoms used in encoding S, and
—$r(I) = \cup_{t=0}^{n} \{holds(l,t) \mid l \text{ is a fluent literal and } l \in s_t\}$.

Then,

(i) The program $\Pi_{formula}(S,I)$ has a unique answer set, X.
(ii) For every temporal formula ϕ such that $formula(n_\phi) \in r(S)$, ϕ is true in I_t, i.e., $I_t \models \phi$, if and only if $hf(n_\phi,t)$ belongs to X where $I_t = \langle s_t, \dots s_n \rangle$.

PROOF. See Appendix A.2 □

Having defined temporal constraints and specified when they are satisfied, adding temporal knowledge to a planning problem in answer set planning is easy. We must encode the knowledge as a temporal formula[12] and then add the set of rules representing this formula and the rules (22)-(33) to Π. Finally, we need to add the

[12]A set of temporal formulae can be viewed as a conjunction of temporal formulae.

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 (Defined as in Sub-section 3.4),
—the rules (22)-(33)
—the rules encoding ϕ and the constraint $\bot \leftarrow not\ hf(n_\phi, 0)$.

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

THEOREM 4.10. 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 $s_0 a_0 \dots a_{n-1} s_n$ is a trajectory achieving Δ and $I \models \phi$ where $I = \langle s_0, \dots, s_n \rangle$, then there exists an answer set M of Π_n^{TLP} such that
  (1) $occ(a_i, i) \in M$ for $i \in \{0, \dots, n-1\}$,
  (2) $s_i = s_i(M)$ for $i \in \{0, \dots, n\}$, 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) $s_0(M) a_0 \dots a_{k-1} s_k(M)$ is a trajectory achieving Δ where $occ(a_i, i) \in M$ for $0 \leq i < k$ and
  (2) $I \models \phi$ where $I = \langle s_0(M), \dots, s_n(M) \rangle$.

PROOF. Observe that the set of literals of the program Π_n, $lit(\Pi_n)$, is a splitting set of the program Π_n^{TPL} and $\Pi_n = b_{lit(\Pi_n)}(\Pi_n^{TPL})$. Thus, M is an answer set of Π_n^{TPL} iff $M = X \cup Y$ where X is an answer set of Π_n and Y is an answer set of $e_{lit(\Pi_n)}(\Pi_n^{TPL} \setminus \Pi_n, X)$ which consists of the set of rules (23)-(33), the set of atoms $\{hf(l,t) \mid holds(l,t) \in X\}$, the rules encoding ϕ, and the constraint $\bot \leftarrow not\ hf(n_\phi, 0)$. This constraint implies that $hf(n_\phi, 0)$ must belong to every answer set M of Π_n^{TPL}.

We now prove (i). It follows from Theorem 3.2 that there exists an answer set X of Π_n such that the first two conditions are satisfied. Because $I \models \phi$, we can apply Theorem 4.9 to show that any answer set Y of $e_{lit(\Pi_n)}(\Pi_n^{TPL} \setminus \Pi_n, X)$ contains $hf(n_\phi, 0)$. Thus, $X \cup Y$ is an answer set satisfying (i).

To prove (ii), it is enough to notice that the answer set X of Π_n, constructed in the proof of Lemma A.4, can be used to construct an answer set M of Π_n^{TPL} such that M satisfies (ii). □

The above theorem shows how control knowledge represented as goal-independent temporal formulae can be exploited in answer set planning. We will now extend this result to allow control knowledge expressed using goal-dependent temporal formulae. Based on Definition 4.8, where satisfaction of goal-dependent temporal formulae is defined, we will need to encode $\Delta \models \psi$ where Δ is the goal and $\mathbf{goal}(\psi)$ is a formula occurring in a control knowledge that we wish to use. To simplify this encoding we make the same assumption that is made in most classical planning literature including [Bacchus and Kabanza 2000]: the goal Δ in a planning problem

$\langle D, \Gamma, \Delta \rangle$ is 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}, the set of atoms $\{fomula(n_{goal_l}) \mid \mathbf{goal}(l)$ is a goal formula occurring in $\phi\}$, and the set of rules

$$hf(n_{goal_f}, T) \leftarrow time(T) \tag{34}$$

for each $f \in \Delta$. Intuitively, these rules assert that f is a part of the goal Δ. The next theorem is about the correctness of $\Pi_n^{TLP+Goal}$.

THEOREM 4.11. For a planning problem $\langle D, \Gamma, \Delta \rangle$ with a consistent action theory (D, Γ) and a temporal formula ϕ,

(i) if $s_0 a_0 \dots a_{n-1} s_n$ is a trajectory achieving Δ and $\langle I, \Delta \rangle \models \phi$ where $I = \langle s_0, \dots, s_n \rangle$, then there exists an answer set M of $\Pi_n^{TLP+Goal}$ such that
  (1) $occ(a_i, i) \in M$ for $i \in \{0, \dots, n-1\}$,
  (2) $s_i = s_i(M)$ for $i \in \{0, \dots, n\}$, 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) $s_0(M) a_0 \dots a_{k-1} s_k(M)$ 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 $I = \langle s_0(M), \dots, s_n(M) \rangle$.

PROOF. To prove this theorem, we first need to modify Theorem 4.9 by (i) allowing goal-dependent formulae to be in the set S; (ii) adding a goal Δ and the rule (34) to the program Π of Theorem 4.9. The proof of this modified theorem is very similar to the proof of Theorem 4.9. We omit it here for brevity. This result, together with Theorem 3.2, proves the conclusion of this theorem. □

4.2 Procedural Knowledge

Procedural knowledge can be thought of as an under-specified sketch of the plans to be generated. The language constructs that we use in this paper to describe procedural knowledge are inspired by GOLOG, an Algol-like logic programming language for agent programming, control and execution; and based on a situation calculus theory of action [Levesque et al. 1997]. GOLOG has primarily been used as a programming language for high-level agent control in dynamical environments (see e.g. [Burgard et al. 1998]). Although a planner can itself be written as a GOLOG program (See Chapter 10 of [Reiter 2000]), in this paper, we view a GOLOG program as an incompletely specified plan (or as 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 in the following (terminating) state 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*. A program is built from *complex actions* and *procedures*. Complex actions, procedures, and programs are constructed using variables, actions, formulae, and procedural program constructs such as **sequence**, **if-then-else**, **while-do**, or **choice**, etc. They are defined as follows.

Definition 4.12. A *complex action* δ with a sequence of variables $X_1, \ldots, X_n$ is

—*a basic complex action*:
  —an expression of the form $a(\sigma_1, \ldots, \sigma_m)$ where a is an m-ary action name, σ_i is either a variable or a constant of the type t_i, and if σ_i is a variable then it belongs to $\{X_1, \ldots, X_n\}$ or
  —an expression of the form ϕ where ϕ is a formula whose free variables are from $\{X_1, \ldots, X_n\}$;
—*a sequence*: an expression of the form $\delta_1; \delta_2$ where δ_1 and δ_2 are complex actions whose free variables are from $\{X_1, \ldots, X_n\}$;
—*a choice of actions*: an expression of the form $\delta_1 \mid \ldots \mid \delta_k$ where δ_j's are complex actions whose free variables are from $\{X_1, \ldots, X_n\}$;
—*a if-then-else*: an expression of the form **if** ϕ **then** δ_1 **else** δ_2 where ϕ is a formula and δ_1 and δ_2 are complex actions whose free variables are from $\{X_1, \ldots, X_n\}$;
—*a while-do*: an expression of the form **while** ϕ **do** δ_1 where ϕ is a formula and δ_1 is a complex action whose free variables are from $\{X_1, \ldots, X_n\}$;
—*a choice of arguments*: an expression of the form $\mathbf{pick}(Y, \{c_1, \ldots, c_n\}, \delta_1)$ where $Y \notin \{X_1, \ldots, X_n\}$, $\{c_1, \ldots, c_n\}$ is a set of constants, and δ_1 is a complex action whose free variables are from $\{X_1, \ldots, X_n, Y\}$; and
—*a procedure call*: an expression of the form $p(X_1, \ldots, X_n)$ where p is a procedure name whose variables are $X_1, \ldots, X_n$.

Definition 4.13. A *procedure* with the name p and a sequence of variables $X_1, \ldots, X_n$ is of the form $(p(X_1, \ldots, X_n) : \delta)$ where δ, called the *body*, is a complex action whose free variables are from $\{X_1, \ldots, X_n\}$.

A procedure $(p(X_1, \ldots, X_n) : \delta)$ is called a *nested procedure* if δ is a procedure call.

Intuitively, a complex action δ represents a sketch of a plan whose variations are given by its variables and its structure. The execution of δ is done recursively over its structure and starts with the instantiation of $X_1, \ldots, X_n$ with some constants $c_1, \ldots, c_n$. In the process, an action might be executed, a formula might be evaluated, other complex actions or procedures might be instantiated and executed. In other words, the execution of δ might depend on the execution of other complex actions. Let δ be a complex action with variables $(X_1, \ldots, X_n)$ and $c_1, \ldots, c_n$ be constants. In the following, we define

—the ground instance of δ with respect to the substitution $\{X_1/c_1, \ldots, X_n/c_n\}$, denoted by $\delta(c_1, \ldots, c_n)$, and

—the set of complex actions that the execution of $\delta(c_1, \ldots, c_n)$ might depend on, denoted by $prim(\delta(c_1, \ldots, c_n))$.

$\delta(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n))$ are defined recursively as follows:

—if $\delta = a(\sigma_1, \ldots, \sigma_m)$, then $\delta(c_1, \ldots, c_n)$ is the action $a(c'_1, \ldots, c'_m)$ where $c'_i = c_j$ if σ_i is the variable X_j and $c'_i = \sigma_i$ if σ_i is a constant and $prim(\delta(c_1, \ldots, c_n)) = \{a(c'_1, \ldots, c'_m)\}$,

—if $\delta = \phi$ then $\delta(c_1, \ldots, c_n) = \phi(c_1, \ldots, c_n)$ where $\phi(c_1, \ldots, c_n)$ is obtained from ϕ by simultaneously replacing every free occurrence of X_i in ϕ by c_i and $prim(\delta(c_1, \ldots, c_n)) = \{\phi(c_1, \ldots, c_n)\}$,

—if $\delta = \delta_1; \delta_2$ then $\delta(c_1, \ldots, c_n) = \delta_1(c_1, \ldots, c_n); \delta_2(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n)) = prim(\delta_1(c_1, \ldots, c_n)) \cup prim(\delta_2(c_1, \ldots, c_n))$,

—if $\delta = \delta_1 \mid \ldots \mid \delta_k$ then $\delta(c_1, \ldots, c_n) = \delta_1(c_1, \ldots, c_n) \mid \ldots \mid \delta_k(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n)) = \bigcup_{i=1}^{k} prim(\delta_i(c_1, \ldots, c_n))$,

—if $\delta =$ **if** ϕ **then** δ_1 **else** δ_2 then $\delta(c_1, \ldots, c_n) =$ **if** $\phi(c_1, \ldots, c_n)$ **then** $\delta_1(c_1, \ldots, c_n)$ **else** $\delta_2(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n)) = \{\phi(c_1, \ldots, c_n)\} \cup prim(\delta_1(c_1, \ldots, c_n)) \cup prim(\delta_2(c_1, \ldots, c_n))$,

—if $\delta =$ **while** ϕ **do** δ_1 then $\delta(c_1, \ldots, c_n) =$ **while** $\phi(c_1, \ldots, c_n)$ **do** $\delta_1(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n)) = \{\phi(c_1, \ldots, c_n)\} \cup prim(\delta_2(c_1, \ldots, c_n))$,

—if $\delta =$ **pick**$(Y, \{y_1, \ldots, y_m\}, \delta_1)$ then $\delta(c_1, \ldots, c_n) = \delta_1(c_1, \ldots, c_n, y_j)$ for some j, $1 \leq j \leq m$, and $prim(\delta(c_1, \ldots, c_n)) = \{prim(\delta_1(c_1, \ldots, c_n, y_j)) \mid j = 1, \ldots, m\}$; and

—If $\delta = p(X_1, \ldots, X_n)$ where $(p(X_1, \ldots, X_n) : \delta_1)$ is a procedure then $\delta(c_1, \ldots, c_n) = \delta_1(c_1, \ldots, c_n)$ and $prim(\delta(c_1, \ldots, c_n)) = \{p(c_1, \ldots, c_n)\} \cup prim(\delta(c_1, \ldots, c_n))$.

A ground instance of a procedure $(p(X_1, \ldots, X_n) : \delta)$ is of the form $(p(c_1, \ldots, c_n) : \delta(c_1, \ldots, c_n))$ where $c_1, \ldots, c_n$ are constants and $\delta(c_1, \ldots, c_n)$ is a ground instance of δ.

In what follows, $\delta(c_1, \ldots, c_n)$ (resp. $(p(c_1, \ldots, c_n) : \delta(c_1, \ldots, c_n))$) will be referred to as a ground complex action (resp. ground procedure). As with complex actions, for a procedure $(p(X_1, \ldots, X_n) : \delta)$ and the constants $c_1, \ldots, c_n$, we define the set of actions that the execution of $p(c_1, \ldots, c_n)$ might depend on by $prim(p(c_1, \ldots, c_n)) = prim(\delta(c_1, \ldots, c_n))$. It is easy to see that under the above definitions, a procedure p may depend on itself. For example, for two procedures "$(p :$ **while** ϕ_1 **do** $q)$" and "$(q :$ **while** ϕ_2 **do** $p)$", we have that $prim(p) = \{p, q, \phi_1, \phi_2\}$ and $prim(q) = \{p, q, \phi_1, \phi_2\}$. Intuitively, this will mean that the execution of p (and q) might be infinite. Since our goal is to use programs, represented as a set of procedures and a ground complex action, to construct plans of finite length, procedures that depend on themselves will not be helpful. For this reason, we define a notion called *well-defined* procedures and limit ourselves to this type of procedure hereafter. We say that a procedure p with variables $X_1, \ldots, X_n$ is *well-defined* if there exists no sequence of constants $c_1, \ldots, c_n$ such that $p(c_1, \ldots, c_n) \in prim(p(c_1, \ldots, c_n))$. We will limit ourselves to sets of procedures in which no two procedures have the same

name and every procedure is well-defined and is not a nested procedure. We call such a set of procedures *coherent* and define programs as follows.

Definition 4.14 *Program.* A *program* is a pair (S, δ) where S is a coherent set of procedures and δ is a ground instantiation of a complex action.

We illustrate the above definition with the following example.

EXAMPLE 4.15. In this example, we introduce the elevator domain from [Levesque et al. 1997] which we use in our initial experiments (Section 4.4). The set of constants in this domain consists of integers between 0 and k representing the floor numbers controlled by the elevator. 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. Alternatively, in the terminology of GOLOG, we can say that the following set of procedures, together with the ground complex action *control*, can be used to control the elevator, so as to satisfy service requests – indicated by the light being on – at different floors. That is, the program for controlling the

elevator is $(S, control)$ where

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

Observe that the formula $\exists N.d(N)\ [on(N)]$, as discussed before, is the shorthand of the disjunction

$$\mathbf{or}(on(0), \mathbf{or}(on(1), \ldots, \mathbf{or}(on(k-1), on(k))))$$

where $0, \ldots, k$ are the floor constants of the domain. □

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* (S, δ). Intuitively, if α is a trace of a program (S, δ) then that means $a_0, \ldots, a_{n-1}$ is a sequence of actions (and α is a corresponding trajectory) that is consistent with the sketch provided by the complex action δ of the program (S, δ) starting from the initial state s_0. Alternatively, it can be thought of as the program (S, δ) *unfolding* 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 4.16 *Trace.* Let $p = (S, \delta)$ 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:

—$\delta = a$ and a is an action, $n = 1$ and $a_0 = a$;
—$\delta = \phi$, $n = 0$ and ϕ holds in s_0;
—$\delta = \delta_1; \delta_2$, and there exists an i such that $s_0 a_0 \ldots s_i$ is a trace of (S, δ_1) and $s_i a_i \ldots s_n$ is a trace of (S, δ_2);
—$\delta = \delta_1 \mid \ldots \mid \delta_n$, and $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of (S, δ_i) for some i;
—$\delta = \mathbf{if}\ \ \phi\ \mathbf{then}\ \ \delta_1\ \mathbf{else}\ \ \delta_2$, and $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of (S, δ_1) if ϕ holds in s_0 or $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of (S, δ_2) if $\neg\phi$ holds in s_0;
—$\delta = \mathbf{while}\ \ \phi\ \mathbf{do}\ \ \delta_1$, $n = 0$ 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 (S, δ_1) and $s_i a_i \ldots s_n$ is a trace of (S, δ); or
—$\delta = \mathbf{pick}(Y, \{y_1, \ldots, y_m\}, \delta_1)$ and $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trace of $(S, \delta_1(y_j))$ for some j, $1 \leq j \leq m$.
—$\delta = p(c_1, \ldots, c_n)$ where $(p(X_1, \ldots, X_n) : \delta_1)$ is a procedure, and
$s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trace of $(S, \delta_1(c_1, \ldots, c_n))$.

□

The above definition allows us to determine whether a trajectory α constitutes a trace of a program (S, δ). This process is done recursively over the structure of δ. More precisely, if δ is not an action or a formula, checking whether α is a trace of (S, δ) amounts to determining whether α is a trace of (S, δ') for some component δ' of δ. We note that because δ is grounded, δ' is also a ground complex action; thus, guaranteeing that (S, δ') is a program and hence the applicability of the definition.

It is easy to see that δ' belongs to $prim(\delta)$. Because of the coherency of S and the finiteness of the domains, this process will eventually stop.

We will now present the **smodels** encoding for programs. The encoding of a program (S, δ) will include the encoding of all procedures in S and the encoding of δ. The encoding of a complex action or a procedure consists of the encoding of all of its ground instances. Similar to the encoding of formulae, each complex action δ will be assigned a distinguished name, denoted by n_δ, whenever it is necessary. Because procedure names are unique in a program, we assign the name $p(c_1, \ldots, c_n)$ to the complex action $\delta(c_1, \ldots, c_n)$ where $(p(X_1, \ldots, X_n) : \delta)$ is a procedure and $c_1, \ldots, c_n$ is a sequence of constants. In other words, $n_{\delta(c_1,\ldots,c_n)} = p(c_1, \ldots, c_n)$. We note that since the body of a procedure is not a procedure call, this will not cause any inconsistency in the naming of complex actions. We now describe the set of rules encoding a complex action δ, denoted by $r(\delta)$, which is defined inductively over the structure of δ as as follows:

—For $\delta = a$ or $\delta = \phi$, $r(\delta)$ is the action a or the rules encoding ϕ, respectively.
—For $\delta = \delta_1; \delta_2$, $r(\delta) = \{sequence(n_\delta, n_{\delta_1}, n_{\delta_2})\} \cup r(\delta_1) \cup r(\delta_2)$.
—For $\delta = \delta_1 \mid \delta_2 \ldots \mid \delta_n$, $r(\delta) = \bigcup_{i=1,\ldots,n} r(\delta_i) \cup \{in(n_{\delta_i}, n_\delta) | i = 1, \ldots, n\} \cup \{choiceAction(n_\delta)\}$.
—For $\delta =$ **if** ϕ **then** δ_1 **else** δ_2, $r(\delta) = r(\phi) \cup r(\delta_1) \cup r(\delta_2) \cup \{if(n_\delta, n_\phi, n_{\delta_1}, n_{\delta_2})\}$.
—For $\delta =$ **while** ϕ **do** δ_1, $r(\delta) = r(\phi) \cup r(\delta_1) \cup \{while(n_\delta, n_\phi, n_{\delta_1})\}$.
—For $\delta =$ **pick**$(Y, \{y_1, \ldots, y_m\}, \delta_1)$,

$$r(\delta) = \bigcup_{j=1}^{j=m} r(\delta_1(y_j)) \cup R$$

where $R = \{choiceArgs(n_\delta, n_{\delta_1(y_j)}) \mid j = 1, \ldots, m)\}$.
—For $\delta = p(c_1, \ldots, c_n)$ where $(p(X_1, \ldots, X_n) : \delta_1)$ is a procedure, $r(\delta) = \{\delta\}$.

A procedure $(p(X_1, \ldots, X_n) : \delta_1)$ is encoded by the set of rules encoding the collection of its ground instances. The encoding of a program (S, δ) consists of $r(\delta)$ and the rules encoding the procedures in S. Observe that because of S's coherence, the set of rules encoding a program (S, δ) is uniquely determined.

Example 4.17. In this example we present the encoding of the program $(S, control)$ from Example 4.15.

We start with the set of rules encoding the ground procedure $(go\_floor(i) : currentFloor(i) \mid up(i) \mid down(i))$ where i is a floor constant. First, we assign the name $go\_floor(i)$ to the complex action $currentFloor(i) \mid up(i) \mid down(i)$ and encode this complex action by the set $r(go\_floor(i))$. This set consists of the following facts:

$$choiceAction(go\_floor(i)).$$
$$in(currentFloor(i), go\_floor(i)).$$
$$in(up(i), go\_floor(i)).$$
$$in(down(i), go\_floor(i)).$$

Similar atoms are needed to encode other instances of the procedure $go\_floor(N)$.

For each floor i, the following facts encode the complex action $go\_floor(i), turnoff(i), open, close$, which is the body of a ground instance of the procedure $(serve(N) : go\_floor(N), turnoff(N), open, close)$:

$$sequence(serve(i), go\_floor(i), serve\_tail\_1(i)).$$
$$sequence(serve\_tail\_1(i), turnoff(i), open\_close).$$
$$sequence(open\_close, open, close).$$

To encode the procedure $(serve\_a\_floor : \mathbf{pick}(N, \{0, 1, \ldots, k\}, (on(N); serve(N)))$, we need the set of rules which encode $serve(i)$, $0 \leq i \leq k$, (above) and the rules encode the complex action $on(i); serve(i)$ for every i. These rules are:

$$sequence(body\_serve\_a\_floor(i), on(i), serve(i)),$$

where $body\_serve\_a\_floor(i)$ is the name assigned to the complex action $on(i); serve(i)$, and the following rule:

$$choiceArgs(serve\_a\_floor, body\_serve\_a\_floor(i)).$$

The following facts encode the procedure $(park$: **if** $currentFloor(0)$ **then** $open$ **else** $[down; park])$:

$$if(park, currentFloor(0), open, park\_1).$$
$$sequence(park\_1, down(0), open).$$

Finally, the encoding of the procedure *control* consists of the rules encoding the formula

$$\mathbf{or}(on(0), \mathbf{or}(on(1), \ldots, \mathbf{or}(on(k-1), on(k))))$$

which is assigned the name '*existOn*' and the following rules:

$$sequence(control, while\_service\_needed, park).$$
$$while(while\_service\_needed, existOn, serve\_a\_floor).$$

□

We now present the AnsProlog rules that realize the operational semantics of programs. We define a predicate $trans(P, T_1, T_2)$ where P is a program and T_1 and T_2 are two time points, $T_1 \leq T_2$. Intuitively, we would like to have $trans(p, t_1, t_2)$ be true in an answer set M iff $s_{t_1}(M)a_{t_1} \ldots a_{t_2-1}s_{t_2}(M)$ is a trace of the program p[13].

$$trans(A, T, T+1) \leftarrow action(A), occ(A, T). \quad (35)$$
$$trans(F, T_1, T_1) \leftarrow formula(F), hf(F, T_1). \quad (36)$$
$$trans(P, T_1, T_2) \leftarrow sequence(P, P_1, P_2), T1 \leq T' \leq T_2, \quad (37)$$
$$trans(P_1, T_1, T'), trans(P_2, T', T_2). \quad (38)$$
$$trans(N, T_1, T_2) \leftarrow choiceAction(N), \quad (39)$$
$$in(P_1, N), trans(P_1, T_1, T_2).$$
$$trans(I, T_1, T_2) \leftarrow if(I, F, P_1, P_2), hf(F, T_1), trans(P_1, T_1, T_2). \quad (40)$$
$$trans(I, T_1, T_2) \leftarrow if(I, F, P_1, P_2), not\ hf(F, T_1), trans(P_2, T_1, T_2). \quad (41)$$

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

$$trans(W, T_1, T_2) \leftarrow while(W, F, P), hf(F, T_1), T_1 < T' \leq T_2, trans(P, T_1, T'), trans(W, T', T_2). \quad (42)$$

$$trans(W, T, T) \leftarrow while(W, F, P), not\ hf(F, T). \quad (43)$$

$$trans(S, T_1, T_2) \leftarrow choiceArgs(S, P), trans(P, T_1, T_2). \quad (44)$$

$$trans(\mathbf{null}, T, T) \leftarrow \quad (45)$$

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 (35) and (36) are for programs consisting of an action and a fluent formula respectively. The other rules are for the remaining cases. For instance, the rule (42) 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 (43) 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.

Observe that we do not have specific rules for complex actions which are procedure calls. This is because of every trace of a procedure call $p(c_1, \ldots, c_n)$, where $p(X_1, \ldots, X_n) : \delta)$ is a procedure, is a trace of the complex action $\delta(c_1, \ldots, c_n)$ — whose name is $p(c_1, \ldots, c_n)$, as described earlier — and vice versa. Furthermore, δ is not a procedure call , traces of δ_1 can be computed using the above rules. The correctness of the above set of rules (see Theorem 4.18) means that procedure calls are treated correctly in our implementation.

To specify that a plan of length n starting from an initial state must obey the sketch specified by a program $p = (S, \delta)$, all we need to do is to add the rules encoding p and the constraint $\leftarrow not\ trans(n_p, 0, n)$ to Π_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 AnsProlog encoding. Let Π_n^{Golog} be the program obtained from Π_n by (i) adding the rules (35)-(45) and (22)-(33), (ii) adding $r(p)$, and (iii) replacing the goal constraint with $\bot \leftarrow not\ trans(n_p, 0, n)$. The following theorem is similar to Theorem 3.2.

THEOREM 4.18. Let (D, Γ) be a consistent action theory and $p = (S, \delta)$ be a program. Then,

(i) for every answer set M of Π_n^{Golog} 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^{Golog} 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. See Appendix A.3 □

To do planning using procedural constraints all we need to do is to add the goal constraint to Π_n^{Golog}, 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.

4.3 HTN-Based Knowledge

The programs in the previous subsections 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 [De Giacomo et al. 2000], 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 [Sacerdoti 1974]. The partial-ordering information allowed in HTN descriptions serves the purpose. Thus all we need is to add constraints 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 to go further than the initial attempt in [Baral and Son 1999] 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 notion. Let $\Sigma = \{(p_1 : \delta_1), \ldots, (p_k : \delta_k)\}$ be a set of procedures with free variables $\{X_1, \ldots, X_n\}$.

—An ordering constraint over Σ has the form $p_i \prec p_j$ where $p_i \neq p_j$.

—A truth constraint is of the form (p_i, ϕ), (ϕ, p_i), or (p_i, ϕ, p_t), where ϕ is a formula whose free variables are from the set $\{X_1, \ldots, X_n\}$.

Given a set of procedures Σ and a set of constraints C over Σ, the execution of Σ will begin with the grounding of Σ and C, i.e., the variables $X_1, \ldots, X_n$ are substituted by some constants $c_1, \ldots, c_n$. The constraints in C stipulate an order in which the procedures in Σ is executed. The intuition behind these types of constraints is as follows:

—An ordering constraint $p_i \prec p_j$ requires that the procedure p_i has to be executed before the procedure p_j.

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

—A constraint of the form (p_i, ϕ, p_t) indicates that ϕ must hold immediately after p_i is executed until p_t begins its execution.

Because a constraint of the form (p_i, ϕ, p_t) implicitly requires that p_i is executed before p_t, for convenience, we will assume hereafter that whenever (p_i, ϕ, p_t) belongs to C, so does $p_i \prec p_t$.

The definition of general complex actions follows.

Definition 4.19 *General Complex Action.* For an action theory (D, Γ), a general complex action with variables $X_1, \ldots, X_n$ is either

—a complex action (Definition 4.12); or
—a pair (Σ, C) where Σ is a set of procedures and C is a set of constraints over Σ and the variables of each procedure in Σ are from $X_1, \ldots, X_n$.

The definition of a procedure or program does not change. The notion of ground instantiation, dependency, and well-definedness of a procedure can be extended straightforwardly to general programs. We will continue to assume that programs in consideration are well-defined. As in the case of programs, the operational semantics of general programs is defined using the notion of trace. In the next definition, we extend the notion of a *trace* to cover the case of general programs.

Definition 4.20 *Trace of general programs.* Let $p = (S, \delta)$ be a general program. We say that a trajectory $s_0 a_0 \ldots a_{n-1} s_n$ is a trace of p if one of the following conditions is satisfied:

—$s_0 a_0 \ldots a_{n-1} s_n$ and (S, δ) satisfy one of the condition in Definition 4.16; or
—$\delta = (\Sigma, C)$, $\Sigma = \{(p_1 : \delta_1), \ldots, (p_k, \delta_k)\}$, 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 $\alpha_1 = s_0 a_0 \ldots s_{j_1}$, $\alpha_2 = s_{j_1} a_{j_1} \ldots s_{j_2}$, …, $\alpha_k = s_{j_{k-1}} a_{j_{k-1}} \ldots s_n$ satisfies the following conditions:
  (1) for each l, $1 \leq l \leq k$, α_l is a trace of (S, δ_{i_l}),
  (2) if $n_t \prec n_l \in C$ then $i_t < i_l$,
  (3) 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
  (4) 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:

$$\underbrace{s_0 a_0 s_1 \ldots a_{j_1-1} s_{j_1}}_{\substack{\alpha_1 \\ \uparrow\downarrow \\ \text{trace of } (S,\delta_{i_1})}} \underbrace{s_{j_1} a_{j_1} \ldots a_{j_2-1} s_{j_2}}_{\substack{\alpha_2 \\ \uparrow\downarrow \\ \text{trace of } (S,\delta_{i_2})}} \ldots \underbrace{s_{j_{l-1}} a_{j_{l-1}+1} \ldots a_{j_l-1} s_{j_l}}_{\substack{\alpha_l \\ \uparrow\downarrow \\ \text{trace of } (S,\delta_{i_l})}} \ldots \underbrace{s_{j_{k-1}} a_{j_{k-1}+1} \ldots a_{j_k-1} s_{j_k}}_{\substack{\alpha_k \\ \uparrow\downarrow \\ \text{trace of } (S,\delta_{i_k})}}$$

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 constraints. 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 '-' denote 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 4.21. 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

$$C = \left\{ \begin{array}{lcl} o & : & move(b,c) \prec move(a,b), \\ f_1 & : & (clear(b), move(b,c)), \\ f_2 & : & (clear(c), move(b,c)), \\ f_3 & : & (clear(b), move(a,b)), \\ f_4 & : & (clear(a), move(a,b)) \end{array} \right\}$$

The constant preceding the semicolon is the name assigned to the formula of C. The encoding of $p = (S, C)$ is as follows:

—$r(p) = \{htn(p, n_S, n_C)\} \cup r(S) \cup r(C)$;
—$r(S) = \{set(n_S), in(move(a,b), n_S), in(move(b,c), n_S)\}$; and
—$r(C)$ consists of
  —the facts defining n_C and declaring its elements

$$\{set(n_C), in(o, n_C), in(f_1, n_C), in(f_2, n_C), in(f_3, n_C), in(f_4, n_C)\}$$

  —the facts defining each of the constraints in C:
    —the order constraint o: $order(o, move(b,c), move(a,b))$,
    —the precondition constraints $f_1, \ldots, f_4$:
      —$precondition(f_1, clear(b), move(b,c))$,
      —$precondition(f_2, clear(c), move(b,c))$,
      —$precondition(f_3, clear(b), move(a,b))$, and
      —$precondition(f_4, clear(a), move(a,b))$.

□

We now present the AnsProlog rules that realize the operational semantics of general programs. For this purpose we need the rules (35)-(45) and the rules for checking the satisfiability of formulae that were 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:

$$trans(N, T_1, T_2) \leftarrow htn(N, S, C), not\ nok(N, T_1, T_2). \tag{46}$$

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 AnsProlog 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., $B_1 < T \leq E_1$ and $B_2 < T \leq E_2$ 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 exactly once during the time (S, C) is executed, we use the following rules:

$$1\{begin(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \quad (47)$$

$$1\{end(N, I, T_3, T_1, T_2) : between(T_3, T_1, T_2)\}1 \leftarrow htn(N, S, C), in(I, S), trans(N, T_1, T_2). \quad (48)$$

The first (resp. second) rule says that I – a program belonging to S – must start (resp. end) its execution exactly once between T_1 and T_2. Here, we use cardinality constraints with variables [Niemelä et al. 1999] 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 shorthand 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 $overlap(.,.,.,.)$.

$$\begin{aligned} used(N,T,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I,S), \\ & begin(N,I,B,T_1,T_2), \\ & end(N,I,E,T_1,T_2), \\ & B \leq T \leq E. \end{aligned} \tag{49}$$

$$not\_used(N,T,T_1,T_2) \leftarrow not\ used(N,T,T_1,T_2). \tag{50}$$

$$\begin{aligned} overlap(N,T,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I_1,S), \\ & begin(N,I_1,B_1,T_1,T_2), \\ & end(N,I_1,E_1,T_1,T_2), \\ & in(I_2,S), begin(N,I_2,B_2,T_1,T_2), \\ & end(N,I_2,E_2,T_1,T_2), \\ & B_1 < T \leq E_1, B_2 < T \leq E_2, I_1 \neq I_2. \end{aligned} \tag{51}$$

The rule (49) 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 (50) 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 make *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 violation of these two conditions is encoded by the following rules which are added to Π.

$$\begin{aligned} nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I,S), T_3 > T_4, \\ & begin(N,I,T_3,T_1,T_2), \\ & end(N,I,T_4,T_1,T_2). \end{aligned} \tag{52}$$

$$\begin{aligned} nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I,S), T_3 \leq T_4, \\ & begin(N,I,T_3,T_1,T_2), \\ & end(N,I,T_4,T_1,T_2), \\ & not\ trans(I,T_3,T_4). \end{aligned} \tag{53}$$

$$\begin{aligned} nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), T_1 \leq T \leq T_2, \\ & not\_used(N,T,T_1,T_2). \end{aligned} \tag{54}$$

$$\begin{aligned} nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), T_1 \leq T \leq T_2, \\ & overlap(N,T,T_1,T_2). \end{aligned} \tag{55}$$

Together the rules (47)-(55) define when the permutation determined by the set of atoms of the form $begin(N,I,B,T_1,T_2)$ and $end(N,I,E,T_1,T_2)$ violates the initial part of condition 8 of Definition 4.20. The rules (47)-(48) require each general program in N to have a unique start and finish time and the rule (52) encodes the

violation when the finish time is earlier than the start time. The rule (53) encodes the violation when the trace of a general program in N does not correspond to its start and finish times. The rule (54) 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 (55) encodes the violation when the trace of two general programs in N overlap.

The next group of rules encode the violation of conditions 8(b) – 8(d) of Definition 4.20.

$$\begin{aligned}
nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I_1,S), begin(N,I_1,B_1,T_1,T_2), \qquad (56)\\
& in(I_2,S), begin(N,I_2,B_2,T_1,T_2),\\
& in(O,C), order(O,I_1,I_2), B_1 > B_2.\\
nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I_1,S), end(N,I_1,E_1,T_1,T_2), \qquad (57)\\
& in(I_2,S), begin(N,I_2,B_2,T_1,T_2), E_1 < T_3 < B_2,\\
& in(O,C), maintain(O,F,I_1,I_2), not\ hf(F,T3).\\
nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I,S), begin(N,I,B,T_1,T_2), \qquad (58)\\
& in(O,C), precondition(O,F,I), not\ hf(F,B).\\
nok(N,T_1,T_2) \leftarrow\ & htn(N,S,C), in(I,S), end(N,I,E,T_1,T_2), \qquad (59)\\
& in(O,C), postcondition(O,F,I), not\ hf(F,E).
\end{aligned}$$

The rule (56) 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 (57) 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 (58) and (59) 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 AnsProlog encoding. For an action theory (D, Γ) and a general program p, let Π_n^{HTN} be the AnsProlog program obtained from Π_n by (i) adding the rules (35)-(45) and (46)-(59), (ii) adding $r(p)$, and (iii) replacing the goal constraint with $\bot \leftarrow not\ trans(n_P, 0, n)$. The following theorem extends Theorem 4.18.

THEOREM 4.22. 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. 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 [Bacchus et al. 2000]. In particular, we tested our program with the Miconic-10 elevator domain. We also tested our program with the Block domain. In both domains, we conducted tests with procedural 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 answer set planning framework [Tuan and Baral 2001]; and (iii) we are not aware of any empirical results indicating the utility of procedural knowledge in planning, especially in answer set planning. (Note that [Reiter 2000] concentrates on using GOLOG to do planning in domains with incomplete information, not on exploiting procedural knowledge in planning.)

We report the results obtained from our experiment with the elevator example from [Levesque et al. 1997] (elp1-elp3) and the Miconic-10 elevator domain (s1-0,. . . ,s5-0), proposed by Schindler Lifts Ltd. for the AIPS 2000 competition [Bacchus et al. 2000]. Note that some of the planners, that competed in AIPS 2000, were unable to solve this problem. The domain description for this example is described earlier in Example 4.15 and the **smodels** code can be downloaded from `http://www.cs.nmsu.edu/~tson/ASPlan/Knowledge`. We use a direct encoding of procedural knowledge in the Block domain to avoid the grounding problem. For this reason, we do not include the results of our experiments on the Block domain in this paper. The results and the encodings of this domain are available on the above mentioned web site. We note that the direct encoding of procedural knowledge in the Block domain yields significantly better results, both in terms of the time needed to find a trajectory as well as the number of problems that can be solved. Whether or not the methodology used in the Block domain can be generalized and applied to other domains is an interesting question that we would like to investigate in the near future.

The initial state for the elevator 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 [Levesque

et al. 1997], which we earlier discussed in Example 4.15.

$$
\begin{aligned}
(go\_floor(N) &: currentFloor(N)|up(N)|down(N)) \\
(serve(N) &: go\_floor(N); turnoff(N); open; close) \\
(serve\_a\_floor &: \mathbf{pick}(N, d(N), (on(N); serve(N)))) \\
(park &: \mathbf{if}\ \ currentFloor(0)\ \mathbf{then}\ \ open\ \mathbf{else}\ \ [down(0); open]) \\
(control &: [\mathbf{while}\ \ \exists N.\{0,1,\ldots,k\}\ [on(N)]\ \mathbf{do}\ \ serve\_a\_floor(N)]; park)
\end{aligned}
$$

We ran experiments on a Sony VAIO laptop with 256 MB Ram and an Intel Pentium 4 1.59 GHz processor, using **lparse** version 1.00.13 (Windows, build Aug 12, 2003) and **smodels** version 2.27. 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	8	2	6	0.380	**0.140**
elp2	12	3	6	0.901	**0.230**
elp3	16	4	6	2.183	**1.381**
elp4	20	5	6	4.566	79.995
s1-0	4	1	2	0.120	**0.020**
s2-0	7	2	4	1.201	**0.080**
s3-0	9	3	6	5.537	**0.310**
s4-0	15	4	8	64.271	**15.211**
s5-0	19	5	10	260.183	1181.158

As can be seen, the encoding with control knowledge yields substantially better performance in situations where the plan length is longer. 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 [Bacchus et al. 2000] (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. We showed that the addition of these three kinds of control knowledge only involves adding a few more rules to a planner written in AnsProlog 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 illustrates the reduction in planning time when procedural domain knowledge is used and the plan length is big.

In the past, temporal domain knowledge has been used in planning in [Bacchus and Kabanza 2000; Doherty and Kvarnstom 1999]. 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 is much better[14] than our implementation using AnsProlog. 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 proofs of our planners and stress the ease of adding the control knowledge to the 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 [Nau et al. 1999]. Unlike our approach in this paper, these planners cannot be separated into 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 [Nau et al. 1999] 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 [Nau et al. 1999] 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 each of the different kinds of domain knowledge separately, the declarative nature 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 such a case, planning will involve finding an action sequence that follows the sketch dictated by the general program and at the same time obeys the temporal domain knowledge. This distinguishes our work from other related work [Huang et al. 1999; Kautz and Selman 1998b; Baral and Son 1999; McIlraith 2000] where the domain knowledge allowed was much more restricted.

A byproduct of the way we deal with procedural knowledge is that, in a propositional environment, our approach to planning with procedural knowledge can be viewed as an off-line interpreter for a GOLOG program. Because of the declarative

[14]This provides a challenge to the community developing AnsProlog* systems to develop AnsProlog* systems that can match or come close to (if not surpass) the performance of planners with procedural knowledge.

nature of AnsProlog the correctness of this interpreter is easier to prove than the earlier interpreters which were mostly written in Prolog.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tocl/2006-V-N/p1-URLend`.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their detailed comments and suggestions that have helped us to improve the paper in several ways. 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 0220590. The work of Sheila McIlraith was also supported by NSERC. A preliminary version of this paper appeared in [Son et al. 2001].

REFERENCES

Babovich, Y. and Lifschitz, V. Computing Answer Sets Using Program Completion.

Bacchus, F. and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence 116,* 1,2, 123–191.

Bacchus, F., Kautz, H., Smith, D., Long, D., Geffner, H., and Koehler, J. 2000. AIPS-00 Planning Competition. In *The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems.*

Baral, C. 2003. *Knowledge Representation, reasoning, and declarative problem solving with Answer sets.* Cambridge University Press, Cambridge, MA.

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

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

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

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

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

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

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence*, 333–377.

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

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

Dimopoulos, Y., Nebel, B., and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of European Conference on Planning.* 169–181.

Doherty, P. and Kvarnstom, J. 1999. 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)*.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. 2000. Planning under incomplete information. In *Proceedings of the First International Conference on Computational Logic (CL'00)*. Springer Verlag, LNAI 1861, 807–821.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. 1998. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*. 406–417.

Erol, K., Nau, D., and Subrahmanian, V. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence 76,* 1-2, 75–88.

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

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

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

Gelfond, M. and Lifschitz, V. 1993. Representing actions and change by logic programs. *Journal of Logic Programming 17,* 2,3,4, 301–323.

Gelfond, M. and Lifschitz, V. 1998. Action languages. *ETAI 3,* 6.

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

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

Kautz, H., McAllester, D., and Selman, B. 1994. Encoding plans in propositional logic. In *Proceedings of KR 94*. 374–384.

Kautz, H. and Selman, B. 1992. Planning as satisfiability. In *Proceedings of ECAI-92*. 359–363.

Kautz, H. and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*. AAAI Press, 1194–1199.

Kautz, H. and Selman, B. 1998a. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop Planning as Combinatorial Search, AIPS-98, Pittsburgh*.

Kautz, H. and Selman, B. 1998b. 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*.

Levesque, H., Reiter, R., Lesperance, Y., Lin, F., and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31,* 1-3 (April-June), 59–84.

Lifschitz, V. 1999a. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, 357–373.

Lifschitz, V. 1999b. Answer set planning. In *Proceedings of ICLP*. 23–37.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1–2, 39–54.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of the Eleventh International Conf. on Logic Programming*, P. Van Hentenryck, Ed. 23–38.

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

Lin, F. 1995. Embracing causality in specifying the indirect effects of actions. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, CA, 1985–1993.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*.

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

McCain, N. and Turner, H. 1995. A causal theory of ramifications and qualifications. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, CA, 1978–1984.

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

Nau, D., Cao, Y., Lotem, A., and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the 16th International Conference on Artificial Intelligence*. AAAI Press, 968–973.

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

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

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence 13,* 1,2, 81–132.

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

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

Sagonas, K., Swift, T., and Warren, D. 1994. XSB as an efficient deductive database engine. In *Proceedings of the SIGMOD*. 442 – 453.

Simons, P., Niemelä, N., and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence 138,* 1–2, 181–234.

Son, T., Baral, C., and McIlraith, S. 2001. Domain dependent knowledge in planning - an answer set planning approach. In *Proceedings of the 6th International Conference on Logic Programming and NonMonotonic Reasoning*. Vienna, 226–239.

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

Tuan, L. and Baral, C. 2001. 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"*. 110–115.

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

Turner, H. 2002. Polynomial-length planning spans the polynomial hierarchy. In *Proc. of Eighth European Conf. on Logics in Artificial Intelligence (JELIA'02)*, pp. 111-124, 2002.

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

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Domain-Dependent Knowledge in Answer Set Planning

TRAN CAO SON
New Mexico State University
CHITTA BARAL and NAM TRAN
Arizona State University
and
SHEILA MCILRAITH
University of Toronto

ACM Transactions on Computational Logic, Vol. V, No. N, August 2006, Pages 1–App–26.

A. APPENDIX A — PROOFS

We apply the Splitting Theorem and Splitting Sequence Theorem [Lifschitz and Turner 1994] several times in our proof[15]. For ease of reading, the basic notation 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. By $lit(\pi)$, we denote the set of literals of a program π.

Appendix A.1 - Proof of Theorem 3.2

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

$$\pi = \Pi_n(D, \Gamma, \Delta) \setminus \{(20),\ \bot \leftarrow \mathit{not\ goal}\},$$

i.e., π is obtained from $\Pi_n(D, \Gamma, \Delta)$ by removing the rules encoding Δ and the constraint $\bot \leftarrow \mathit{not\ goal}$. Let S be a set of literals of the form $holds(l, t)$. Abusing the notation, we say that S is *consistent with respect to* $\mathbf{F}$ (or consistent, for short) if for every pair of a fluent $f \in \mathbf{F}$ and a time moment t, S does not contain both $holds(f, t)$ and $holds(\neg f, t)$. For a set of causal laws K and a set of fluent literals Y, let

$$M_K(Y) = Y \cup \{l \mid \exists\, \mathbf{caused}(\{p_1, \ldots, p_k\}, l) \in K \ \ s.t.\ \ Y \models p_1 \wedge \ldots \wedge p_k\}. \quad (60)$$

[15]To be more precise, we use a modified version of these theorems since programs in this paper contain constraints of the form (6) which were not discussed in [Lifschitz and Turner 1994]. The modification is discussed in Appendix B.

It is easy to see that $M_K(Y)$ is a monotonic function and bounded above by $\mathbf{F}$. Thus, the sequence $\langle M_K^i(Y)\rangle_{i<\infty}$, where $M_K^0(Y) = Y$ and $M_K^{i+1}(Y) = M_K(M_K^i(Y))$ for $i \geq 0$, is a convergent sequence. In other words, there exists some $i \geq 0$ such that $M_K^{j+1}(Y) = M_K^j(Y)$ for $j \geq i$. Let $C(K,Y) = \bigcup_{i<\infty} M_K^i(Y)$. We have that $C(K,Y)$ is closed under K. We can prove the following lemma.

LEMMA A.1. *For a set of causal laws K and a set of fluent literals Y,*

(1) $C(K,Y)$ is consistent and $C(K,Y) = Cl_K(Y)$ iff $Cl_K(Y)$ is defined; and
(2) $C(K,Y)$ is inconsistent iff $Cl_K(Y)$ is undefined.

PROOF. The lemma is trivial for inconsistent set of literals Y. We need to prove it for the case where Y is consistent.

(1) Consider the case where $C(K,Y)$ is consistent. From the definition of the function M_K in (60), we have that $M_K(S) \subseteq Cl_K(S)$ for every set of literals S. Therefore, $M_K^i(Y) \subseteq Cl_K(Y)$ for every i. Thus, $C(K,Y) \subseteq Cl_K(Y)$. Because $C(K,Y)$ is closed under K and $Cl_K(Y)$ is the least set of literals closed under K, we can conclude that $Cl_K(Y) \subseteq C(K,Y)$.

    To complete the proof of the first item, we need to show that if $Cl_K(Y)$ is defined, then $C(K,Y)$ is consistent and $C(K,Y) = Cl_K(Y)$. Again, it follows immediately from Equation (60) that $M_K(S) \subseteq Cl_K(S)$ for every consistent set S. This implies that $M_K^i(Y) \subseteq Cl_K(Y)$ for every i. Thus, we have that $C(K,Y) \subseteq Cl_K(Y)$. This implies the consistency of $C(K,Y)$. The equality $C(K,Y) = Cl_K(Y)$ follows from the closeness of $C(K,Y)$ with respect to K and the definition of $Cl_K(Y)$. This concludes the first item of the lemma.

(2) Consider the case where $C(K,Y)$ is inconsistent. Assume that $Cl_K(Y)$ is defined. By definition of $M_K^i(Y)$, we know that if $M_K^i(Y)$ is inconsistent then $M_K^{i+1}(Y)$ is also inconsistent. Thus, there exists an integer k such that $M_K^k(Y)$ is consistent and $M_K^{k+1}(Y)$ is inconsistent. Because of the consistency of $Cl_K(Y)$ and $M_K^k(Y) \subseteq Cl_K(Y)$, we conclude that $M_K^{k+1}(Y) \setminus Cl_K(Y) \neq \emptyset$. Consider $l \in M_K^{k+1}(Y) \setminus Cl_K(Y)$. By the definition of M_K, there exists some static causal law

$$\mathbf{caused}(\{p_1,\ldots,p_n\}, l)$$

    in K such that $\{p_1,\ldots,p_n\} \subseteq M_K^k(Y)$. This implies that $Cl_K(Y)$ is not closed under K, which contradicts the definition of $Cl_K(Y)$. This shows that $Cl_K(Y)$ is undefined.

    Now, consider the case where $Cl_K(Y)$ is undefined. Using contradiction and the result of the first item, we can also show that $C(K,Y)$ is inconsistent. This concludes the proof of the second item of the lemma.

□

LEMMA A.2. *For a set of causal laws K and a set of fluent literals Y, for every integer k, the program consisting of the following rules:*

$$\begin{array}{rcll} holds(l,k) & \leftarrow & holds(l_1,k),\ldots,holds(l_m,k) & (\textit{if } \mathbf{caused}(\{l_1,\ldots,l_m\},l) \in K) \\ holds(l,k) & \leftarrow & & (\textit{if } l \in Y) \\ \bot & \leftarrow & fluent(f), holds(f,k), holds(\neg f,k) & (\textit{if } f \in \mathbf{F}) \end{array}$$

(1) has a unique answer set $\{holds(l,k) \mid l \in Cl_K(Y)\}$ *iff* $Cl_K(Y)$ *is defined; and*
(2) does not have an answer set iff $Cl_K(Y)$ *is undefined.*

PROOF. Let us denote the given program by Q and P be the program consisting of the rules of Q with the head different than $\bot$. Since P is a positive program, we know that it has a unique answer set, says X. It is easy to see that if X is consistent with respect to $\mathbf{F}$, then X is the unique answer set of Q; otherwise, Q does not have an answer set.

It is easy to see that $X = \{holds(l,k) \mid l \in C(K,Y)\}$ is the unique answer set of P. Consider the two cases:

(1) $Cl_K(Y)$ is defined. Lemma A.1 implies that $C(K,Y) = Cl_K(Y)$, and hence, X is consistent with respect to $\mathbf{F}$. This implies that X is the unique answer set of Q.
(2) $Cl_K(Y)$ is undefined. Lemma A.1 implies that $C(K,Y)$ is inconsistent, which implies that X is inconsistent with respect to $\mathbf{F}$, and hence, Q does not have an answer set.

□

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. First, we simplify π by using the splitting theorem [Lifschitz and Turner 1994] (Theorem B.1, Appendix B). Let V be the set of literals in the language of π whose parameter list does not contain the time parameter, i.e., V consists of auxiliary atoms of the form $literal(l)$, $fluent(f)$, $action(a)$, $contrary(l_1,l_2)$.

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 rules that define actions, fluents, and the rules (14)–(17). 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:

$$holds(l,t{+}1) \leftarrow occ(a,t), holds(l_1,t), \ldots, holds(l_k,t). \quad (61)$$
(if $\mathbf{causes}(a,l,\{l_1,\ldots,l_k\}) \in D$)

$$holds(l,t) \leftarrow holds(l_1,t), \ldots, holds(l_m,t). \quad (62)$$
(if $\mathbf{caused}(\{l_1,\ldots,l_m\},l) \in D$)

$$possible(a,t) \leftarrow holds(l_1,t), \ldots, holds(l_t,t). \quad (63)$$
(if $\mathbf{executable}(a,\{l_1,\ldots,l_t\}) \in D$)

$$holds(l,0) \leftarrow \quad (64)$$
(if $\mathbf{initially}(l) \in \Gamma$)

$$occ(a,t) \leftarrow possible(a,t), not\ nocc(a,t). \quad (65)$$
(if a is an action)

$$nocc(a,t) \leftarrow occ(b,t). \quad (66)$$

$$\begin{aligned}
&&&\text{(for every pair of actions } a \neq b)\\
holds(f,t{+}1) &\leftarrow holds(f,t), not\ holds(\neg f,t{+}1). &&(67)\\
&&&\text{(for every fluent } f)\\
holds(\neg f,t{+}1) &\leftarrow holds(\neg f,t), not\ holds(f,t{+}1). &&(68)\\
&&&\text{(for every fluent } f)\\
\bot &\leftarrow holds(f,t), holds(\neg f,t). &&(69)\\
&&&\text{(for every fluent } f)
\end{aligned}$$

It follows from the splitting theorem that to prove the consistency and correctness of π it is enough to prove the consistency of π_1 and that π_1 correctly implements the transition function Φ of D. We prove this in the next lemmas.

LEMMA A.3. *Let (D,Γ) be a consistent action theory. Let X be an answer set of π_1. Then,*

(1) $s_t(X)$ is a state of D for every $t \in \{0,\ldots,n\}$ [16] *for every $t \in \{0,\ldots,n-1\}$,*

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

(3) for every $t \in \{0,\ldots,n-1\}$, 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

$$\begin{aligned}
U_t \;=\; &\{holds(l,k) \mid l \text{ is a literal and } k \leq t\}\cup\\
&\{occ(a,k) \mid a \text{ is an action and } k \leq t\}\cup\\
&\{nocc(a,k) \mid a \text{ is an action and } k \leq t\}\cup\\
&\{possible(a,k) \mid a \text{ is an action and } k \leq t\} \cup \{\bot\},
\end{aligned}$$

is a splitting sequence of π_1. Since X is an answer set of π_1, by the splitting sequence theorem (Theorem B.2, 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 $b_{U_0}(\pi_1)$ and
—for every $t > 0$, X_t is an answer set of $e_{U_t}(b_{U_t}(\pi_1) \setminus b_{U_{t-1}}(\pi_1), \bigcup_{i\leq t-1} X_i)$.

We will prove the lemma by inductively proving that for every t, $0 \leq t \leq n$, the following holds:

(i) 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) 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 $s_{t-1}(X_{t-1}) = s_t(X_t)$.

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

Base case: $t = 0$. Trivially, X_0 satisfies (iv). So, we only need to show that X_0 satisfies (i)-(iii). Let $P_0 = b_{U_0}(\pi_1)$. We have that P_0 consists of only rules of the form (62)-(66) and (69) with $t = 0$. 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 $holds(l, 0)$ 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 $occ(a, 0) \in X_0$ iff $occ(a, 0) \in N_0$. Hence, to prove that X_0 satisfies (i)-(iii), 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 (64) and (62). Because of the consistency of (D, Γ), we have that $s_0 = \{f \mid \mathbf{initially}(f) \in \Gamma\}$ is consistent and hence $\{holds(f, 0) \mid \mathbf{initially}(f) \in \Gamma\}$ is consistent with respect to $\mathbf{F}$. It follows from Lemma A.2 that $M_0 = \{holds(f, 0) \mid f \in s_0\}$ is the unique answer set of $b_{Z_0}(P_0)$ 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

$$e_{Z_0,M_0} = \begin{cases} possible(a,0) \leftarrow & (a1) \\ \quad \text{(if } \mathbf{executable}(a, \{l_1, \ldots, l_m\}) \in D & \\ \quad \text{and } holds(l_i, 0) \in M_0) & \\ occ(a,0) \leftarrow possible(a,0), not\ nocc(a,0). & (a2) \\ nocc(a,0) \leftarrow occ(b,0). & (a3) \\ \quad \text{(for every pair of actions } a \neq b) & \\ \leftarrow holds(f,0), holds(\neg f,0) & (a4) \\ \quad \text{(for every fluent } f) & \end{cases}$$

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.

Inductive 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:

$$holds(l, k) \leftarrow \tag{70}$$

$$
\begin{aligned}
& && (\text{if } occ(a, k-1) \in M_{k-1}, \\
& && \mathbf{causes}(a, l, \{l_1, \ldots, l_m\}) \in D, \\
& && holds(l_i, k-1) \in M_{k-1}) \\
holds(l, k) & \leftarrow && holds(l_1, k), \ldots, holds(l_m, k). && (71) \\
& && (\text{if } \mathbf{caused}(\{l_1, \ldots, l_m\}, l) \in D) \\
possible(a, k) & \leftarrow && holds(l_1, k), \ldots, holds(l_t, k). && (72) \\
& && (\text{if } \mathbf{executable}(a, \{l_1, \ldots, l_t\}) \in D) \\
occ(a, k) & \leftarrow && possible(a, k), not\ nocc(a, k). && (73) \\
& && (\text{if } a \text{ is an action}) \\
nocc(a, k) & \leftarrow && occ(b, k). && (74) \\
& && (\text{for every pair of actions } a \neq b) \\
holds(f, k) & \leftarrow && not\ holds(\neg f, k). && (75) \\
& && (\text{if } holds(f, k-1) \in M_{k-1}) \\
holds(\neg f, k) & \leftarrow && not\ holds(f, k). && (76) \\
& && (\text{if } holds(\neg f, k-1) \in M_{k-1}) \\
\bot & \leftarrow && holds(f, k), holds(\neg f, k). && (77)
\end{aligned}
$$

From the constraint (77), 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 (75) 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 (76), 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. We now prove the other items of the conclusion. 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:

$$
\begin{array}{rll}
holds(l,k) & \leftarrow & \qquad (b1)\\
 & & (\text{if } occ(a,k-1) \in M_{k-1}, \mathbf{causes}(a,l,\{l_1,\ldots,l_m\}) \in D,\\
 & & \ holds(l_i,k-1) \in M_{k-1} \text{ for } i=1,\ldots,m)\\
holds(l,k) & \leftarrow & holds(l_1,k),\ldots,holds(l_n,k). \qquad (b2)\\
 & & (\text{if } \mathbf{caused}(\{l_1,\ldots,l_n\},l) \in D)\\
holds(f,k) & \leftarrow & \qquad (b3)\\
 & & (\text{ if } holds(f,k-1) \in M_{k-1} \text{ and } holds(\neg f,k) \notin Y_k)\\
holds(\neg f,k) & \leftarrow & \qquad (b4)\\
 & & (\text{ if } holds(\neg f,k-1) \in M_{k-1} \text{ and } holds(f,k) \notin Y_k)\\
\bot & \leftarrow & holds(f,k), holds(\neg f,k). \qquad (b5)\\
 & & (\text{for every fluent } f)
\end{array}
$$

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 $C_2 = \{l \mid holds(l,k) \in Q_2\}$. There are two cases:

—**Case 1:** M_{k-1} does not contain an atom of the form $occ(a,k-1)$. We have that $Q_1 = \emptyset$ and $C_2 \subseteq s_{k-1}(X_{k-1})$. From Lemma A.2, we know that $(\pi_k)^{Y_k}$ has a unique answer set $\{holds(f,k) \mid f \in Cl_{D_C}(C_2)\}$ which is Y_k. Because $s_k(X_k) = \{f \mid f \in Y_k\}$ and the definition of Cl_{D_C}, we have that $s_k(X_k) \subseteq s_{k-1}(X_{k-1})$. The completeness and consistency of $s_k(X_k)$ and $s_{k-1}(X_{k-1})$ implies 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 $occ(a,k-1) \in M_{k-1}$. Because of the rule (b1) we have that $C_1 = E(a, s_{k-1}(X_{k-1}))$. The completeness of $s_k(X_k)$ and $s_{k-1}(X_{k-1})$ and the rules (b3)-(b4) imply that $C_2 = s_k(X_k) \cap s_{k-1}(X_{k-1})$. Furthermore, Lemma A.2 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, $s_k(X_k) = Cl_{D_C}(E(a, s_{k-1}(X_{k-1})) \cup (s_k(X_k) \cap s_{k-1}(X_{k-1})))$. This implies that $s_k(X_k) \in \Phi(a, s_{k-1}(X_{k-1}))$. 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

$$
e_{Y_k} = \left\{
\begin{array}{lll}
possible(a,k) & \leftarrow & \\
 & & (\text{if } \mathbf{executable}(a,\{l_1,\ldots,l_m\}) \in D\\
 & & \ \text{and } holds(l_i,k) \in Y_k)\\
occ(a,k) & \leftarrow & possible(a,k), not\ nocc(a,k).\\
 & & (\text{if } a \text{ is an action})\\
nocc(a,k) & \leftarrow & occ(b,k).\\
 & & (\text{for every pair of actions } a \neq b)
\end{array}
\right.
$$

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 $occ(a,k)$ or 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 A.4. *For every trajectory $s_0 a_0 \dots a_{n-1} s_n$ in D and a consistent action theory (D,Γ), π_1 has an answer set X such that*

(1) $s_t(X) = s_t$ for every t, $0 \leq t \leq n$, and

(2) $occ(a_t,t) \in X$ for every t, $0 \leq t \leq n-1$.

PROOF. We prove the theorem by constructing an answer set X of π_1 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

$$\begin{array}{rcl} U_t & = & \{holds(l,k) \mid l \text{ is a literal and } k \leq t\} \cup \\ & & \{occ(a,k) \mid a \text{ is an action and } k \leq t\} \cup \\ & & \{nocc(a,k) \mid a \text{ is an action and } k \leq t\} \cup \\ & & \{possible(a,k) \mid a \text{ is an action and } k \leq t\} \cup \{\bot\}. \end{array}$$

For every t, $0 \leq t \leq n$, let $R_t = \{possible(a,t) \mid a \text{ is executable in } s_t\}$. We define a sequence of sets of literals $\langle X_t \rangle_{t=0}^n$ as follows.

—For $0 \leq t \leq n-1$,

$$\begin{array}{rcl} X_t & = & \{holds(f,t) \mid f \in s_t\} \cup \{occ(a_t,t)\} \cup \\ & & \{nooc(b,t) \mid b \text{ is an action in } D,\ b \neq a_t\} \cup R_t. \end{array}$$

—If $R_n \neq \emptyset$, then let a_n be an arbitrary action that is executable in s_n and

$$\begin{array}{rcl} X_n & = & \{holds(f,n) \mid f \in s_n\} \cup \{occ(a_n,n)\} \cup \\ & & \{nooc(b,t) \mid b \text{ is an action in } D,\ b \neq a_n\} \cup R_n. \end{array}$$

—If $R_n = \emptyset$, then

$$X_n = \{holds(f,n) \mid f \in s_n\} \cup \{nooc(b,t) \mid b \text{ is an action in } D\} \cup R_n,$$

We will prove that $\langle X_t \rangle_{t=0}^n$ is a solution to π_1 with respect to $\langle U_t \rangle_{t=0}^n$. This amounts to prove that

—X_0 is an answer set of $b_{U_0}(\pi_1)$ and

—for every $t > 0$, X_t is an answer set of $e_{U_t}(b_{U_t}(\pi_1) \setminus b_{U_{t-1}}(\pi_1), \bigcup_{i \leq t-1} X_i)$.

We first prove that X_0 is an answer set of $P_0 = b_{U_0}(\pi_1)$. By the construction of P_0

and X_0, we have that $(P_0)^{X_0}$ consists of the following rules:

$$(P_0)^{X_0} = \left\{ \begin{array}{rcll} holds(f,0) & \leftarrow & (\text{if } \mathbf{initially}(f) \in \Gamma) & (a1) \\ holds(l,0) & \leftarrow & holds(l_1,0),\ldots,holds(l_m,0). & (a2) \\ & & (\text{if } \mathbf{caused}(\{l_1,\ldots,l_m\},l) \in D) & \\ possible(a,0) & \leftarrow & holds(l_1,0),\ldots,holds(l_m,0). & (a3) \\ & & (\text{if } \mathbf{executable}(a,\{l_1,\ldots,l_m\}) \in D) & \\ occ(a_0,0) & \leftarrow & possible(a_0,0). & (a4) \\ nocc(b,0) & \leftarrow & occ(a,0). & (a5) \\ & & (\text{for every pair of actions } b \neq a) & \\ \bot & \leftarrow & holds(f,0), holds(\neg f,0) & (a6) \\ & & (\text{for every fluent } f) & \end{array} \right.$$

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 of the form (a1). Because of s_0 is closed under the static causal laws in D, we conclude that X_0 is closed under the rule of the form (a2). The definition of R_0 guarantees that X_0 is closed under the rule of the form (a3). Since $s_0 a_0 \ldots a_{n-1} s_n$ 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 of $(P_0)^{X_0}$.

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 $P_t = e_{U_t}(b_{U_{t-1}}(\pi_1) \setminus b_{U_{t-1}}(\pi_1), \bigcup_{i \leq t-1} X_i)$. 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 $M_{k-1} = \bigcup_{i \leq k-1} X_i$. The program P_k consists of the following rules:

$$\begin{array}{rcll} holds(l,k) & \leftarrow & & (78) \\ & & (\text{if } \mathbf{causes}(a_{k-1},l,\{l_1,\ldots,l_m\}) \in D,\ holds(l_i,k-1) \in M_{k-1}) & \\ holds(l,k) & \leftarrow & holds(l_1,k),\ldots,holds(l_m,k). & (79) \end{array}$$

$$\text{(if } \mathbf{caused}(\{l_1,\ldots,l_m\},l)\in D)$$
$$possible(a,k) \leftarrow holds(l_1,k),\ldots,holds(l_t,k). \quad (80)$$
$$\text{(if } \mathbf{executable}(a,\{l_1,\ldots,l_t\})\in D)$$
$$occ(a,k) \leftarrow possible(a,k), not\ nocc(a,k). \quad (81)$$
$$\text{(if } a \text{ is an action)}$$
$$nocc(a,k) \leftarrow occ(b,k). \quad (82)$$
$$\text{(for every pair of actions } a\neq b)$$
$$holds(f,k) \leftarrow not\ holds(\neg f,k). \quad (83)$$
$$\text{(if } holds(f,k-1)\in M_{k-1})$$
$$holds(\neg f,k) \leftarrow not\ holds(f,k). \quad (84)$$
$$\text{(if } holds(\neg f,k-1)\in M_{k-1})$$
$$\bot \leftarrow holds(f,k), holds(\neg f,k). \quad (85)$$

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 (78)-(79) and (83)-(84). 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:

$$holds(l,k) \leftarrow \quad \text{(b1)}$$
$$\text{(if } \mathbf{causes}(a_{k-1},l,\{l_1,\ldots,l_m\})\in D,\ holds(l_i,k-1)\in M_{k-1} \text{ for } i=1,\ldots,m)$$
$$holds(l,k) \leftarrow holds(l_1,k),\ldots,holds(l_n,k). \quad \text{(b2)}$$
$$\text{(if } \mathbf{caused}(\{l_1,\ldots,l_n\},l)\in D)$$
$$holds(f,k) \leftarrow \quad \text{(b3)}$$
$$\text{(if } holds(f,k-1)\in M_{k-1} \text{ and } holds(\neg f,k)\notin Y_k)$$
$$holds(\neg f,k) \leftarrow \quad \text{(b4)}$$
$$\text{(if } holds(\neg f,k-1)\in M_{k-1} \text{ and } holds(f,k)\notin Y_k)$$
$$\bot \leftarrow holds(f,k), holds(\neg f,k). \quad \text{(b5)}$$

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 $C_2 = \{l \mid holds(l,k)\in Q_2\}$. Rule (b1) and the fact that $f \in s_{k-1}(X_{k-1})$ iff $holds(f,k-1)\in M_{k-1}$ imply that $C_1 = E(a_{k-1}, s_{k-1}(X_{k-1}))$. Similar argument allows us to conclude that $C_2 = s_k(X_k)\cap s_{k-1}(X_{k-1})$. Lemma A.2 implies that $(\pi_k)^{Y_k}$ has a unique answer set $Y = \{holds(f,k) \mid f\in Cl_{D_C}(C_1\cup C_2)\}$. Since $occ(a_{k-1},k-1)\in M_{k-1}$ and $s_k(X_k)\in\Phi(a_{k-1},s_{k-1}(X_{k-1}))$, we have that $s_k(X_k) = Cl_{D_C}(C_1\cup C_2)$. It follows from the definition of Y_k that $Y_k = Y$. 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

$$e_{Z_k,Y_k} = \begin{cases} possible(a,k) \leftarrow (\text{if } \mathbf{executable}(a,\{l_1,\ldots,l_m\}) \in D \\ \qquad\qquad\qquad\quad \text{and } holds(l_i,k) \in Y_k) \\ occ(a,k) \leftarrow possible(a,k), not\ nocc(a,k). \\ \qquad\qquad\qquad\quad (\text{if } a \text{ is an action}) \\ nocc(a,k) \leftarrow occ(b,k). \\ \qquad\qquad\qquad\quad (\text{for every pair of actions } a \neq b) \end{cases}$$

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

$$(e_{Z_k,Y_k})^{N_k} = \begin{cases} possible(a,k) \leftarrow (\text{if } \mathbf{executable}(a,\{l_1,\ldots,l_m\}) \in D \\ \qquad\qquad\qquad\quad \text{and } holds(l_i,k) \in Y_k) \\ occ(a_k,k) \leftarrow possible(a_k,k). \\ nocc(a,k) \leftarrow occ(b,k). \\ \qquad\qquad\qquad\quad (\text{for every pair of actions } a \neq b) \end{cases}$$

Let R_k be the set of atoms occurring in the first rule of $(e_{Z_k,Y_k})^{N_k}$. Because $s_0a_0\ldots a_ns_n$ 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_1 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_1. 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, $0 \leq t \leq n$. The lemma is proved. □

The above lemmas lead to the following corollaries.

COROLLARY A.5. *Let X be an answer set of π. Then,*

(i) $s_t(X)$ is a state of D for every t, $0 \leq t \leq n$,
(ii) if X contains $occ(a,t)$ then a is executable in $s_t(X)$ and $s_{t+1}(X) \in \Phi(a, s_t(X))$ for every t, $0 \leq t \leq n-1$, and
(iii) if $occ(a,t) \notin X$ for every action a, then $s_{t+1}(X) = s_t(X)$ for every t, $0 \leq t \leq n-1$.

PROOF. It follows from the splitting theorem that $Y = X \cap lit(\pi_1)$ is an answer set of π_1. Because $s_t(X) = s_t(Y)$ and Lemma A.3, we conclude that X satisfies the (i)-(iii). □

COROLLARY A.6. *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*

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

PROOF. From Lemma A.4, there exists an answer set Y of π_1 such that $s_t(Y) = s_t$ and $occ(a_t,t) \in Y$. Again, from the splitting theorem, we can conclude that there exists an answer set X of π such that $Y = X \cap lit(\pi_1)$. Because $s_t(X) = s_t(Y)$, we conclude that X satisfies (i)-(ii). □

The next observation is also useful.

OBSERVATION A.7. *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 the above corollaries we can prove Theorem 3.2.

THEOREM A.8 (MAIN TEXT 3.2). For a planning problem $\langle D, \Gamma, \Delta \rangle$,

(i) if $s_0 a_0 \dots a_{n-1} s_n$ 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 $i \in \{0, \dots, n-1\}$ and
   (2) $s_i = s_i(M)$ for $i \in \{0, \dots, n\}$.
   and

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

PROOF. We have that $\Pi_n = \pi \cup \{(20), \leftarrow not\ goal\}$. Assume that $\Delta = p_1 \wedge \dots \wedge p_k$.

(i). Since $s_0 a_0 \dots a_{n-1} s_n$ is a trajectory achieving Δ, the existence of X that satisfies the condition (i) of the theorem follows from Corollary A.6. Furthermore, because of $s_n \models \Delta$, we can conclude that $holds(p_i, n) \in X$ for every i, $1 \leq i \leq k$. Thus, $X \cup \{goal\}$ is an answer set of Π_n. This implies the existence of M satisfying (i).

(ii). Let M be an answer set of Π_n. It is easy to see that this happens iff $goal \in M$ and $X = M \setminus \{goal\}$ is an answer set of π and $holds(p_i, n) \in X$ for every i, $1 \leq i \leq k$. It follows from Observation A.7 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 A.5, 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 $s_0(M) a_0 \dots a_{k-1} s_k(M)$ is a trajectory and $s_k(M) = s_n(M)$. Moreover, Δ holds in $s_n(M) = s_k(M)$. Thus, $s_0(M) a_0 \dots a_{k-1} s_k(M)$ is a trajectory achieving Δ. Furthermore, it follows from Corollary A.5 and the rules (65) and (66) that if $k < n$ then M does not contain literals of the form $possible(a, k)$. This implies that no action is executable in $s_k(M)$ if $k < n$. □

Appendix A.2 - Proofs of Theorem 4.9

THEOREM A.9 (MAIN TEXT 4.9). Let S be a finite set of goal-independent temporal formulae, $I = \langle s_0, s_1 \dots s_n \rangle$ be a sequence of states, and

$$\Pi_{formula}(S, I) = \Pi_{formula} \cup r(I) \cup r(S)$$

where

—$r(S)$ is the set of atoms used in encoding S, and
—$r(I) = \cup_{t=0}^{n} \{holds(l, t) \mid l \text{ is a fluent literal and } l \in s_t\}$.

Then,

(i) The program $\Pi_{formula}(S, I)$ has a unique answer set, X.

(ii) For every temporal formula ϕ such that $formula(n_\phi) \in r(S)$, ϕ is true in I_t, i.e., $I_t \models \phi$, if and only if $hf(n_\phi, t)$ belongs to X where $I_t = \langle s_t, \ldots s_n \rangle$.

PROOF. First, we prove (i). We know that if a program is locally stratified then it has a unique answer set. We will show that $\Pi_{formula}(S, I)$ (more precisely, the set of ground rules of $\Pi_{formula}(S, I)$) is indeed locally stratified. To accomplish that we need to find a mapping λ from literals of $\Pi_{formula}(S, I)$ 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 $\Pi_{formula}(S, I)$, then $\lambda(A_0) \geq \lambda(A_i)$ for all $1 \leq i \leq n$ and $\lambda(A_0) > \lambda(B_j)$ for all $1 \leq j \leq m$.

To define λ, we first associate to each constant ϕ that occurs as the first parameter of the predicate $formula(.)$ in $\Pi_{formula}(S, I)$ a non-negative number $\sigma(\phi)$ as follows.

—$\sigma(l) = 0$ if l is a literal (recall that if l is a literal then $n_l = l$).

—$\sigma(n_\phi) = \sigma(n_{\phi_1}) + 1$ if ϕ has the form **negation**(ϕ_1), **next**(ϕ_1), **eventually**(ϕ_1), or **always**(ϕ_1).

—$\sigma(n_\phi) = \max\{\sigma(n_{\phi_1}), \sigma(n_{\phi_2})\} + 1$ if ϕ has the form **and**(ϕ_1, ϕ_2), **or**(ϕ_1, ϕ_2), or **until**(ϕ_1, ϕ_2).

We define λ as follows.

—$\lambda(hf(n_\phi, t)) = 5 * \sigma(n_\phi) + 2$,

—$\lambda(hf\_during(n_\phi, t, t')) = 5 * \sigma(n_\phi) + 4$, and

—$\lambda(l) = 0$ for every other literal of $\Pi_{formula}(S, I)$.

Examining all the rules in $\Pi_{formula}(S, I)$, we can verify that λ has the necessary property.

We now prove (ii). Let X be the answer set of $\Pi_{formula}(S, I)$. We prove the second conclusion of the lemma by induction over $\sigma(n_\phi)$.

Base: Let ϕ be a formula with $\sigma(n_\phi) = 0$. 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 (22), proves the base case.

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

Let ϕ be such a formula that $\sigma(n_\phi) = k + 1$. Because of the definition of σ, ϕ is a non-atomic formula. We have the following cases:

—**Case 1:** $\phi =$ **negation**(ϕ_1). We have that $\sigma(n_{\phi_1}) = \sigma(n_\phi) - 1 = k$. Because of $formula(n_\phi) \in X$ and $negation(n_\phi, n_{\phi_1}) \in X$, $hf(n_\phi, t) \in X$ iif the body of rule (27) is satisfied by X iff $hf(n_{\phi_1}, t) \notin X$ iff $s_t \not\models \phi_1$ (by inductive hypothesis) iff $s_t \models \phi$.

—**Case 2:** $\phi =$ **and**(ϕ_1, ϕ_2). Similar to the first case, it follows from the rule (23) and the facts $formula(n_\phi)$ and $and(n_\phi, n_{\phi_1}, n_{\phi_2})$ that $hf(n_\phi, t) \in X$ iif the body of rule (23) is satisfied by X iff $hf(n_{\phi_1}, t) \in X$ and $hf(n_{\phi_2}, t) \in X$ iff $s_t \models \phi_1$ and $s_t \models \phi_2$ (inductive hypothesis) iff $s_t \models \phi$.

—**Case 3:** $\phi = \mathbf{or}(\phi_1, \phi_2)$. The proof is similar to the above cases, relying on the two rules (25), (26), and the fact $formula(n_\phi) \in X$ and $or(n_\phi, n_{\phi_1}, n_{\phi_2}) \in X$.

—**Case 4:** $\phi = \mathbf{until}(\phi_1, \phi_2)$. We have that $\sigma(n_{\phi_1}) \leq k$ and $\sigma(n_{\phi_2}) \leq k$. Assume that $I_t \models \phi$. By Definition 4.7, 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 (28), 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 (28), 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 5:** $\phi = \mathbf{next}(\phi_1)$. Note that $\sigma(n_{\phi_1}) \leq k$. Rule (31) 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 6:** $\phi = \mathbf{always}(\phi_1)$. We note that $\sigma(n_{\phi_1}) \leq k$. Observe that $hf(n_\phi, t)$ is supported only by rule (29). So $hf(n_\phi, t) \in X$ iff $hf\_during(n_{\phi_1}, t, n) \in X$. 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 7:** $\phi = \mathbf{eventually}(\phi_1)$. We know that $hf(n_\phi, t) \in X$ is supported only by rule (30). 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 $\sigma(n_{\phi_1}) \leq k$, 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.3 - Proof of Theorem 4.18

We first prove some lemmas that are needed for proving Theorem 4.18. Abusing the notation, by π_f we denote the program consisting of the rules of π (Appendix A.1) and the set of rules $\Pi_{formula}$ where the time constant T takes the value between 0 and n.

LEMMA A.10. *For a consistent action theory (D, Γ), a ground complex action p, and an answer set M of Π_n^{Golog} 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 that the union of the set of literals of π_f and the set of rules and atoms encoding p, i.e., $U = lit(\pi_f) \cup r(p)$, is a splitting set of Π_n^{Golog}. Furthermore, $b_U(\Pi_n^{Golog}) = \pi_f \cup r(p)$. Thus, by the splitting theorem, M is an answer set of Π_n^{Golog} iff $M = X \cup Y$ where X is an answer set of $\pi_f \cup r(p)$, and Y is an answer set of $e_U(\Pi_n^{Golog} \setminus \pi_f, X)$. Because of the constraint $\bot \leftarrow not\ trans(n_p, 0, n)$, we know that if M is an answer set of Π_n^{Golog} then every answer set Y of $e_U(\Pi_n^{Golog} \setminus \pi_f, X)$ 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[17]:

[17]Recall that for simplicity, in encoding programs or formulae we use l or a as the name associated

(*) for every ground complex action q with the name n_q and two time points t_1, t_2 such that $q \neq \mathbf{null}$ 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 $\pi_3 = e_U(\Pi_n^{Golog} \setminus \pi_f, X)$. We have that π_3 consists of the following rules:

$$trans(a,t,t+1) \leftarrow (\text{if } action(a) \in X,\ occ(a,t) \in X) \quad (86)$$

$$trans(f,t_1,t_1) \leftarrow (\text{if } formula(f,.) \in X,\ hf(f,t_1) \in X) \quad (87)$$

$$trans(p,t_1,t_2) \leftarrow t1 \leq t' \leq t_2, trans(p_1,t_1,t'), trans(p_2,t',t_2). \quad (88)$$
$$(\text{if } sequence(p,p_1,p_2) \in X))$$

$$trans(n,t_1,t_2) \leftarrow trans(p_1,t_1,t_2). \quad (89)$$
$$(\text{if } choiceAction(n) \in X,\ in(p_1,n) \in X)$$

$$trans(i,t_1,t_2) \leftarrow trans(p_1,t_1,t_2). \quad (90)$$
$$(\text{if } if(i,f,p_1,p_2) \in X,\ hf(f,t_1) \in X)$$

$$trans(i,t_1,t_2) \leftarrow trans(p_2,t_1,t_2). \quad (91)$$
$$(\text{if } if(i,f,p_1,p_2) \in X,\ hf(f,t_1) \notin X)$$

$$trans(w,t_1,t_2) \leftarrow t_1 < t' \leq t_2, trans(p,t_1,t'), trans(w,t',t_2). \quad (92)$$
$$(\text{if } while(w,f,p) \in X,\ hf(f,t_1) \in X)$$

$$trans(w,t,t) \leftarrow (\text{if } while(w,f,p) \in X, hf(f,t) \notin X) \quad (93)$$

$$trans(s,t_1,t_2) \leftarrow trans(p,t_1,t_2). \quad (94)$$
$$(\text{if } choiceArgs(s,p) \in X)$$

$$trans(\mathbf{null},t,t) \leftarrow \quad (95)$$

Clearly, π_3 is a positive program. Thus, the unique answer set of π_3, denoted by Y, is the fix-point of the T_{π_3} operator, defined by $T_{\pi_3}(X) = \{o \mid$ there exists a rule $o \leftarrow o_1, \ldots, o_n$ in π_3 such that $o_i \in X$ for $i = 1, \ldots, n\}$. Let $Y_k = T_{\pi_3}^k(\emptyset)$. By definition $Y = \lim_{n\to\infty} Y_n$.

For every atom $o \in Y$, let $\rho(o)$ denote the smallest integer k such that for all $0 \leq t < k$, $o \notin Y_t$ and for all $t \geq k$, $o \in Y_t$. (Notice that the existence of $\rho(o)$ is guaranteed because T_{π_3} 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_3 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 (86), (87), or (93).

—r is of the form (86). 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 A.5 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.

to l or a, respectively.

—r is of the form (87). Then $q = \phi, t_2 = t_1 = t$, where ϕ is a formula and $hf(n_\phi, t)$ is in X. By Theorem 4.9, ϕ holds in $s_t(X)$. Again, because $s_t(M) = s_t(X)$, we have that $s_t(M)$ is a trace of q.

—r is of the form (93). 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 $T_{\pi_3}(Y_k)$, there is some rule $trans(n_q, t_1, t_2) \leftarrow A_1, \ldots A_m$ in π_3 such that all $A_1, \ldots A_m$ are in Y_k. From the construction of π_3, we have the following cases:

—r is a rule of the form (88). Then, there exists q_1, q_2, t' such that $sequence(n_q, n_{q_1}, n_{q_2}) \in X$, $trans(n_{q_1}, t_1, t') \in Y_k$, 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 $sequence(n_q, nq_1, n_{q_2}) \in X$ we know that $q = q_1; q_2$. By Definition 4.16, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ is a trace of q.

—r is a rule of the form (89). 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 4.16, it is also a trace of q.

—r is a rule of the form (90). Then, by the construction of π_3, 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 4.9, 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 (91). Similarly to the previous items, we know that there exist ϕ, q, q_1, and q_2 such that $if(n_q, n_\phi, n_{q_1}, n_{q_2}) \in X$, $hf(n_\phi, t_1) \notin X$, 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 4.9, $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ 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 (92). 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 4.16, 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 (94). Then, q has the form $\mathbf{pick}(X, \{c_1, \ldots, c_n\}, q_1)$. Therefore, $choiceArgs(n_q, n_{q_1}(c_j))$ is in X for $j = 1, \ldots, n$. $trans(n_q, t_1, t_2) \in Y$ implies that there exists an integer j, $1 \leq j \leq n$, such $trans(n_{q_1(c_j)}, t_1, t_2) \in Y_k$. By the definition of ρ, $\rho(trans(n_{q_1}(c_j), t_1, t_2)) \leq k$. 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 program $q_1(c_j)$. Thus, we can 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. □

To prove the reverse of Lemma A.10, we define a function μ that maps each ground complex action q into an integer $\mu(q)$ that reflects the notion of complexity of q (or the number of nested constructs 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 $q = \mathbf{if}\ \ \phi\ \mathbf{then}\ \ q_1\ \mathbf{else}\ \ q_2$, $\mu(q) = 1 + \mu(q_1) + \mu(q_2)$.
—For $q = q_1 \mid \ldots \mid q_m$, $\mu(q) = 1 + \max\{\mu(q_i) \mid i = 1, \ldots, m\}$.
—For $q = \mathbf{while}\ \ \phi\ \mathbf{do}\ \ q_1$, $\mu(q) = 1 + \mu(q_1)$.
—For $q = \mathbf{pick}(X, \{c_1, \ldots, c_n\}, q_1)$, $\mu(q) = 1 + \max\{\mu(q_1(c_j)) \mid j = 1, \ldots, n\}$.
—For $q = p(c_1, \ldots, c_n)$ where $(p(X_1, \ldots, X_n) : \delta_1)$ is a procedure, $\mu(q) = 1 + \mu(\delta_1(c_1, \ldots, c_n))$.

It is worth noting that $\mu(q)$ is always defined for programs considered in this paper.

LEMMA A.11. *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^{Golog} 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^{Golog} that satisfies the conditions of the lemma. Similar to the proof of Lemma A.10, we split Π_n^{Golog} using $U = lit(\pi_f) \cup r(p)$. This implies that M is an answer set of Π_n^{Golog} iff $M = X \cup Y$ where X is an answer set of $b_U(\Pi_n^{Golog})$ and Y is an answer set of $\pi_3 = e_U(\Pi_n^{Golog} \setminus b_U(\Pi_n^{Golog}), X)$, which is the program consisting of the rules (86)-(95) 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 A.6, we know that π_f 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 $\pi_f \cup r(p)$ 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_3 is a positive program we know that π_3 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^{Golog}. 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 with the name n_q and there exists two integers t_1 and t_2 such that $s_{t_1}(M)a_{t_1} \dots a_{t_2-1}s_{t_2}(M)$ 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 4.16, we have that $t_2 = t_1$. It follows from the assumption that $s_{t_1}(M)$ is a trace of q that $s_{t_1}(M)$ satisfies ϕ. By Theorem 4.9, $hf(n_\phi, t_1) \in X$, and hence, we have that $trans(n_\phi, t_1, t_1) \in Y$ (because of rule (87)).

—$q = a$ where a is an action. Again, by Definition 4.16, 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 (86) of π_3, 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 4.16, there exists t', $t_1 \leq t' \leq t_2$, such that $s_{t_1}a_{t_1} \dots s_{t'}$ is a trace of q_1 and $s_{t'}a_{t'} \dots s_{t_2}$ 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 $sequence(n_q, n_{q_1}, n_{q_2}) \in M$. By rule (88), $trans(n_q, t_1, t_2)$ must be in M.

—$q = q_1 \mid \dots \mid q_i$. Again, by Definition 4.16, $s_{t_1}a_{t_1} \dots a_{t_2-1}s_{t_2}$ 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 (89), $trans(n_q, t_1, t_2)$ is in M.

—$q =$ **if** ϕ **then** q_1 **else** q_2. Consider two cases:

  —ϕ holds in s_{t_1}. This implies that $s_{t_1}(M)a_{t_1} \dots a_{t_2-1}s_{t_2}(M)$ is a trace of q_1. Because of Theorem 4.9, $hf(n_\phi, t_1) \in M$. Since $\mu(q_1) < \mu(q)$, $trans(n_{q_1}, t_1, t_2) \in M$ by inductive hypothesis. Thus, according to rule (90), $trans(n_q, t_1, t_2)$ must belong to M.

  —ϕ does not holds in s_{t_1}. This implies that $s_{t_1}(M)a_{t_1} \dots a_{t_2-1}s_{t_2}(M)$ is a trace of q_2. Because of Theorem 4.9, $hf(n_\phi, t_1)$ 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 (91), $trans(n_q, t_1, t_2) \in M$.

—$q =$ **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 (93), $trans(n_q, t_1, t_2)$ 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 $s_{t_1} a_{t_1} \dots s_{t'}$ is a trace of q_1 and $s_{t'} a_{t'} \dots s_{t_2}$ 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 4.9, $hf(n_\phi, t_1)$ is in M and from the rule (92), $trans(n_q, t_1, t_2)$ is in M.

—$q = \mathbf{pick}(X, \{c_1, \dots, c_n\}, q_1)$. So, there exists an integer j, $1 \leq j \leq n$, such that the trace of q is a trace of $q_1(c_j)$. Since $\mu(q_1(c_j)) < \mu(q)$, we have that $trans(n_{q_1(c_j)}, t_1, t_2) \in M$. This, together with the fact that $choiceArgs(n_q, n_{q_1(c_j)}) \in r(p)$, and the rule (94) imply that $trans(n_q, t_1, t_2)$ is in M.

—$q = p(c_1, \dots, c_n)$ for some procedure $(p(X_1, \dots, X_n), q_1)$. This implies that $s_{t_1}(M) a_{t_1} \dots a_{t_2 - 1} s_{t_2}(M)$ is a trace of $q_1(c_1, \dots, c_n)$. Since $\mu(q_1(c_1, \dots, c_n)) < \mu(q)$, we have that $trans(n_{q_1}, t_1, t_2) \in M$. Since $n_{q_1} = p(c_1, \dots, c_n)$ and $n_q = q$, we have that $trans(n_q, t_1, t_2) \in M$. This proves the inductive step for this case as well.

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

We now prove the Theorem 4.18.

THEOREM A.12 (MAIN TEXT 4.18). Let (D, Γ) be a consistent action theory and p be a program. Then,

(i) for every answer set M of Π_n^{Golog} 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 $s_0 a_0 \dots a_{n-1} s_n$ is a trace of p then there exists an answer set M of Π_n^{Golog} 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. (i) follows from Lemma A.10 and (ii) follows from Lemma A.11. □

Appendix A.4 - Proof of Theorem 4.22

Let p now be a general program. To prove Theorem 4.22, we will extend the Lemmas A.10-A.11 to account for general programs. Similarly to the proofs of Lemmas A.10-A.11, we will split Π_n^{HTN} by the set $U = lit(\pi_f) \cup r(p)$. Thus M is an answer set of Π_n^{HTN} iff $M = X \cup Y$ where X is an answer set of $\pi_f \cup r(p)$ and Y is an answer set of the program $e_U(\Pi_n^{HTN} \setminus b_U(\Pi_n^{HTN}), X)$, denoted by π_4, which consists of the rules of program π_3 (with the difference that a program is now a general program) and the following rules:

$$trans(n, t_1, t_2) \leftarrow not\ nok(n, t_1, t_2). \quad (\text{if } htn(n, s, c) \in X) \tag{96}$$

$$1\{begin(n, i, t_3, t_1, t_2) :$$

$$between(t_3, t_1, t_2)\}1 \leftarrow trans(n, t_1, t_2). \quad (97)$$
$$(\text{if } htn(n, s, c) \in X,\ in(i, s) \in X)$$
$$1\{end(n, i, t_3, t_1, t_2)\ :$$
$$between(t_3, t_1, t_2)\}1 \leftarrow trans(n, t_1, t_2). \quad (98)$$
$$(\text{if } htn(n, s, c) \in X,\ in(i, s) \in X)$$
$$used(n, t, t_1, t_2) \leftarrow begin(n, i, b, t_1, t_2), \quad (99)$$
$$end(n, i, e, t_1, t_2),$$
$$b \leq t \leq e.$$
$$(\text{if } htn(n, s, c) \in X,\ in(i, s) \in X)$$
$$not\_used(n, t, t_1, t_2) \leftarrow not\ used(n, t, t_1, t_2). \quad (100)$$
$$overlap(n, t, t_1, t_2) \leftarrow begin(n, i_1, b_1, t_1, t_2), \quad (101)$$
$$end(n, i_1, e_1, t_1, t_2),$$
$$begin(n, i_2, b_2, t_1, t_2),$$
$$end(n, i_2, e_2, t_1, t_2),$$
$$b_1 < t \leq e_1, b_2 < t \leq e_2.$$
$$(\text{if } htn(n, s, c) \in X,\ in(i_1, s) \in X,\ in(i_2, s) \in X)$$
$$nok(n, t_1, t_2) \leftarrow t_3 > t_4, begin(n, i, t_3, t_1, t_2), \quad (102)$$
$$end(n, i, t_4, t_1, t_2).$$
$$(\text{if } htn(n, s, c) \in X,\ in(i, s) \in X)$$
$$nok(n, t_1, t_2) \leftarrow t_3 \leq t_4, begin(n, i, t_3, t_1, t_2), \quad (103)$$
$$end(n, i, t_4, t_1, t_2),$$
$$not\ trans(i, t_3, t_4).$$
$$(\text{if } htn(n, s, c) \in X,\ in(i, s) \in X)$$
$$nok(n, t_1, t_2) \leftarrow t_1 \leq t \leq t_2, not\_used(n, t, t_1, t_2). \quad (104)$$
$$(\text{if } htn(n, s, c) \in X)$$
$$nok(n, t_1, t_2) \leftarrow t_1 \leq t \leq t_2, overlap(n, t, t_1, t_2). \quad (105)$$
$$(\text{if } htn(n, s, c) \in X)$$
$$nok(n, t_1, t_2) \leftarrow begin(n, i_1, b_1, t_1, t_2), \quad (106)$$
$$begin(n, i_2, b_2, t_1, t_2),$$
$$b_1 > b_2.$$
$$(\text{if } htn(n, s, c) \in X,\ in(i_1, s) \in X,\ in(i_2, s) \in X,$$
$$in(o, c) \in X,\ order(o, i_1, i_2) \in X))$$
$$nok(n, t_1, t_2) \leftarrow end(n, i_1, e_1, t_1, t_2), \quad (107)$$
$$begin(n, i_2, b_2, t_1, t_2), e_1 < t_3 < b_2.$$
$$(\text{if } htn(n, s, c) \in X,\ in(i_1, s) \in X,\ in(i_2, s) \in X,$$
$$in(o, c)\ \in X,\ maintain(o, f, i_1, i_2) \in X,$$
$$\text{and } hf(f, t_3) \notin X)$$

$$nok(n,t_1,t_2) \leftarrow begin(n,i,b,t_1,t_2), end(n,i,e,t_1,t_2), \quad (108)$$
$$(\text{if } htn(n,s,c) \in X,\ in(i,s) \in X,$$
$$in(o,c) \in X, precondition(o,f,i) \in X$$
$$\text{and } hf(f,b) \notin X)$$
$$nok(n,t_1,t_2) \leftarrow begin(n,i,b,t_1,t_2), end(n,i,e,t_1,t_2). \quad (109)$$
$$(\text{if } htn(n,s,c) \in X,\ in(i,s) \in X,$$
$$in(o,c) \in X,\ postcondition(o,f,i) \in X,$$
$$\text{and } hf(f,e) \notin X)$$

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 π_4 has the first parameter as a program[18]. Hence, we can associate $\mu(q)$ to each literal l of π_4 where q is the first parameter of l. 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 to use splitting theorem in our proofs, the following observation is useful.

Observation A.13. *The two cardinality constraint rules (97)-(98) can be replaced by the following normal logic program rules:*

$$begin(n,i,t,t_1,t_2) \leftarrow trans(n,t_1,t_2),$$
$$t_1 \le t \le t_3 \le t_2, not\ nbegin(n,i,t,t_1,t_2).$$
$$nbegin(n,i,t,t_1,t_2) \leftarrow trans(n,t_1,t_2),$$
$$t_1 \le t \le t_2, t_1 \le t_3 \le t_2, t \ne t_3, begin(n,i,t_3,t_1,t_2).$$
$$end(n,i,t,t_1,t_2) \leftarrow trans(n,t_1,t_2),$$
$$t_1 \le t \le t_3 \le t_2, not\ nend(n,i,t,t_1,t_2).$$
$$nend(n,i,t,t_1,t_2) \leftarrow trans(n,t_1,t_2),$$
$$t_1 \le t \le t_2, t_1 \le t_3 \le t_2, t \ne t_3, end(n,i,t_3,t_1,t_2).$$

That is, let π^* be the program obtained from π_4 by replacing the rules (97)-(98) with the above set of rules. Then, M is an answer set of π_4 iff $M' = M \cup \{nbegin(n,i,t_3,t_1,t_2) \mid begin(n,i,t,t_1,t_2) \in M, t \ne t_3, t_1 \le t, t_3 \le t_2\} \cup \{nend(n,i,t_3,t_1,t_2) \mid t \ne t_3,\ t_1 \le t, t_3 \le t_2, end(n,i,t,t_1,t_2) \in M\}$ is an answer set of π^*.

The next lemma generalizes Lemma A.10.

Lemma A.14. *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 π_4), and t_1, t_2 be two time points such that $q \ne$ **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 $M = X \cup Y$.*

[18]More precisely, a program name.

PROOF. Let $U_k = \{l \mid l \in lit(\pi_4),\ \mu(l) \leq k\}$.

From observation A.13, we know that we can use the splitting theorem on π_4. It is easy to see that $\langle U_k \rangle_{k<\infty}$ is a splitting sequence of π_4. From the finiteness of π_4 and the splitting sequence theorem, we have that $Y = \bigcup_{i<\infty} Y_i$ where

(1) Y_0 is an answer set of the program $b_{U_0}(\pi_4)$ and
(2) for every integer i, Y_{i+1} is an answer set for $e_{U_i}(b_{U_{i+1}}(\pi_4) \setminus b_{U_i}(\pi_4), \bigcup_{j\leq i} Y_j)$.

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 $b_{U_0}(\pi_4)$ consists of all the rules of π_4 whose program has level 0. It follows from Lemma A.10 $s_{t_1}(M)a_{t_1}s_{t_1+1}(M)\ldots a_{t_2-1}s_{t_2}(M)$ 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 $e_{U_k}(b_{U_{k+1}}(\pi_4) \setminus b_{U_k}(\pi_4), \bigcup_{j\leq k} Y_k)$ which consists of rules of the form (96)-(109) and (88)-(94) whose program has the level $k + 1$, i.e., $\mu(q) = k + 1$. 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 (88)-(94), the argument is similar to the argument using in the inductive step for the corresponding case in Lemma A.10. Notice a minor difference though: in Lemma A.10, we do not need to use $\mu(q)$.

—r is a rule of the form (96), which implies that $q = (S, C)$ where S is a set of programs and C is a 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 (97) and (98), 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 (103), 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 (99)-(105) 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 4.20.

Consider now an ordering $q_{j_1} \prec q_{j_2}$ in C. This implies that the body of rule (106) 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 4.20.

Similarly, using (107)-(109) we can prove that the permutation $\{j_b \mid q_j \in S\}$ also satisfies the third and fourth conditions of Definition 4.20.

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 A.11.

LEMMA A.15. *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 $lit(\pi_f)\cup r(q)$ 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 π_4 such that $trans(n_p,0,n)\in Y$. Similar to the proof of Lemma A.14, we use $\langle U_k\rangle_{k<\infty}$ as a splitting sequence of π_4 where $U_k=\{u\mid u\in lit(\pi_4),\ \mu(u)\leq k\}$. From the splitting sequence theorem, we have that $Y=\bigcup_{i<\infty}Y_i$ where

(1) Y_0 is an answer set of the program $b_{U_0}(\pi_4)$ and
(2) for every integer i, Y_{i+1} is an answer set for $e_{U_i}(b_{U_{i+1}}(\pi_4)\setminus b_{U_i}(\pi_4),\bigcup_{j\leq i}Y_j)$.

We prove the lemma by induction over $\mu(q)$. Similar to Lemma A.11, we prove this by proving a stronger conclusion:

(*) There exists an answer set $Y=\bigcup_{i<\infty}Y_i$ of π_4 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 A.11 .

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 $\pi^+=e_{U_k}(b_{U_{k+1}}(\pi_4)\setminus b_{U_k}(\pi_4),\bigcup_{j\leq k}Y_k)$ 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 4.20). 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 (88). 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 4.16, $s_{t_1} a_{t_1} \ldots s_{t_2}$ 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 (89)-(94).

—r is a rule of the form (96) 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 (96) too.

—r is a rule of the form (97) and (98). 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 4.20, 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 (97) and (98).

—r is a rule of the form (100)-(109). 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 (99). 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 (88)-(94). 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 (96). 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 (97). 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 (98). 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 (99). 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. □

THEOREM A.16 (MAIN TEXT 4.22). 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 A.14 and (ii) follows from Lemma A.15. □

B. APPENDIX B — SPLITTING THEOREM

In this appendix, we review the basics of the Splitting Theorem [Lifschitz and Turner 1994]. Because programs in this paper do not contain classical negations, some of the definitions have been modified from the original presentation in [Lifschitz and Turner 1994].

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 atoms of $\mathcal{LP}$, A, is a splitting set of Π if for every rule $r \in \Pi$, 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 $lit(r) \subseteq A$.

Given a splitting set A for Π, and a set X of atoms 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;

we create 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 sets of atoms satisfying the following two properties:

(1) X is an answer set of $b_A(\Pi)$; and
(2) Y is an answer set of $e_A(\Pi \setminus b_A(\Pi), X)$;

The splitting set theorem is as follows.

THEOREM B.1 SPLITTING SET THEOREM, [LIFSCHITZ AND TURNER 1994]. *Let A be a splitting set for a program Π. A set S of atoms is a consistent answer set of Π iff $S = X \cup Y$ for some solution $\langle X, Y\rangle$ to Π with respect to A.* □

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 atoms 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 $e_{A_\alpha}(b_{A_{\alpha+1}}(\Pi) \setminus b_{A_\alpha}(\Pi), \bigcup_{\gamma\leq\alpha} E_\gamma)$; and
(3) For any limit ordinal $\alpha < \mu$, $E_\alpha = \emptyset$.

The splitting set theorem is generalized for splitting sequence next.

THEOREM B.2 SPLITTING SEQUENCE THEOREM, [LIFSCHITZ AND TURNER 1994]. *Let $A = \langle A_\alpha\rangle_{\alpha<\mu}$ be a splitting sequence of the program Π. A set of atoms E is an 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.* □

To apply Theorems B.1-B.2 to programs with constraints of the form (6), we need to modify the notation of the bottom of a program relative to a set of atoms as follows.

Let $\Pi = \Pi_1 \cup \Pi_2$ be a program with constrains where Π_1 is a set of rules of the form (5) and Π_2 is a set of rules of the form (6). For a splitting set A of Π, we define $b_A(\Pi) = b_A(\Pi_1) \cup c_A(\Pi_2)$ where $c_A(\Pi_2) = \{r \mid r \in \Pi_2,\ lit(r) \subseteq A\}$.

We can prove that Theorems B.1-B.2 hold for programs with constraints. For example, if A is a splitting of the program Π, then S is an answer set of Π iff $A = X \cup Y$ where X is an answer set of $b_A(\Pi)$ and Y is an answer set of $e_A(\Pi \setminus b_A(\Pi), X)$. The proof of the modified theorems is based on two observations: (i) a set A of atoms from $lit(\Pi)$ is a splitting set of Π iff it is a splitting set of Π_1 (because $\bot \notin lit(\Pi)$); and (ii) a set of atoms S is an answer set of Π iff S is answer set of Π_1 and S satisfies the rules of Π_2.

Received July 2002; revised December 2003; accepted April 2004