Relating logic programming theories of actions and partial order planning

Chitta Baral
Department of Computer Science
University of Texas at El Paso
El Paso, Texas 79968, U.S.A.
chitta@cs.utep.edu
915-747-6952/5030 (phone/fax)

Abstract

In this paper we argue that logic programming theories of action allow us to identify subclasses when the corresponding logic program has nice properties (such as acyclicity) that guarantees decidability and sometimes polynomial time plan testing. As an example we extend the action description language $\mathcal{A}$ to allow executability conditions and show its formalization in logic programming. We show the relationship between the execution of partial order planners and the SLDNF tree with respect to the corresponding logic programs. In the end we briefly discuss how this relationship helps us in extending partial order planners to extended languages by following the corresponding logic program.

1 Introduction and Motivation

The aim of this paper is to bridge the gap between theories of reasoning about actions and planning implementations.

Research in theories of actions is concerned with developing formal theories that allows us to express and reason with various facets such as inertia (and the associated frame problem [GL93, Bro87]), hypothetical facts [GL93, BGP95] constraints and indirect effects (and the associated qualification and ramification problem [LR94, KL94, Bar95]), non-deterministic and other complex effects of actions [Bar95, Pin94], narratives and actual situations [MS94, PR93, BGP95], concurrent and compound actions[LLL+94, LS92, BG93], causality and dependency between fluents[MT95, Lin95, Bar95, GL95], knowledge producing or sensing actions [SL93], etc.

One of the agenda behind the research in reasoning about actions has been to contribute towards the development of 'autonomous agents' that can 'perform' in a dynamic environment. To 'perform' in a dynamic environment the agents have to sense, reason, *plan*, and execute actions (hopefully according to the generated plan).

Planning means finding a sequence of actions which if executed will achieve a given set of goals. A simple approach to planning would be to (non-deterministically) guess a sequence of actions and verify (i.e. *test*) that the sequence of actions when executed indeed achieves the goal. In a deterministic computer the guessing part is implemented by a searching strategy. More sophisticated approaches involve pushing the testing deeper into the searching (or into a *generate* function) and/or testing with respect to a set of sequences of actions. This results in various kinds of planners such as the partial order planners [Wel94, KKY95], a planner that uses [BF95] plan graphs, etc. The intractability of planning in the STRIPS (augmented with conditional effects) domain and its subdomains is analyzed in detail in [ENS92, ENS95]. *This inherent complexity has led researchers developing planners to concentrate on domains where testing is "easy".* Until recently this meant the domain of choice was STRIPS and recently some planners consider ADL which is more expressive than STRIPS but still has the testing part tractable.[1]

But to our knowledge very few planners have used a more expressive language than ADL. In other words recent advances in the theory of reasoning about actions has had very little impact in developing planners for the extended languages and features formalized in these theories. We believe there are *two main reasons* for this.

The *first* reason is that the proposed theories have not paid much attention to the "easy testing" criteria that is a de-facto requirement in most planning implementations. This is because most planning implementations already have to deal with the complexity of the searching part and hence, tend to prefer action theories where at least plan verification is easy.

The *second* reason is that the entailment relation corresponding to theories of actions tell us about testing and plan existence (not plan construction, which requires an answer extraction procedure) and in recent planners (such as the POP planners) the testing is so much integrated into the generate part (for efficiency purposes) that connections between these planners and theories of actions is hidden. This has led to questions in the mind of many researchers in both areas. Some of the common questions are: (i) how is the frame problem taken care of in a POP, (ii) how can I test if a sequence of actions is a plan using a POP, (iii) how do I use a theory of reasoning about actions in a POP if an explicit regression operator is not provided, etc.

Our goal in this paper is to contribute towards eliminating these two obstacles.

We tackle the first obstacle by using logic programming theories of actions. By logic programming theories of actions we refer to those theories of actions that either use logic programming directly to formalize some aspects of reasoning about actions [AB90, BG94] or those that provide translations to logic programs [GL93, BG93, BGP95, DDS93, HT93]. Besides expressibility [Bar95] [2] the main reasons we prefer[3] logic programming theory of actions are:

[1]In case of ADL it is shown in [Ped94] that the regression operators can be constructed easily, i.e. in polynomial time.

[2]The non-classical implication in logic programs and recent extensions such as epistemic disjunction make logic programming highly expressive. [BG94] has a discussion on this expressibility and its usefulness in knowledge representation.

[3]There are some similar efforts [Lin95] that use classical logic formalization of theories of actions and

(a) for most theories of actions we can identify subclasses where the logic programming translation has the nice property of 'acyclicity' [AB91] where entailment is decidable in general, and identify further subclasses where testing is polynomial time, and
(b) theoretical results [AP94] linking logic programming entailment to PROLOG derivation enable us (in many cases) [BGP95] not only to prove correctness of logic programs but also to prove correctness of corresponding PROLOG programs[4].

We tackle the second obstacle by showing the correspondence between testing in a logic program and in a partial order planner. In particular we extend the action description language $\mathcal{A}$ [GL93] to include executability conditions and schema variables. The resultant language is more expressive than STRIPS. We then provide several translations of descriptions in this extended language to logic programs. We then show the correspondence between doing testing with respect to these logic programs and with respect to the planners. These correspondence suggests how planning methodologies can be extended to larger domains by using the extended domains considered in the various theories of reasoning about actions. Finally, we briefly discuss some of these extensions.

2 A Simple theory of actions

In this section we consider a high level action description language (that extends the language $\mathcal{A}$ [GL93] by allowing executability conditions) and its translation to logic programs. Later we describe the standard partial order planner (POP) and show how the planner can be extended as we extend the language.

2.1 Syntax of $\mathcal{A}_1$

A description of an action domain in the language $\mathcal{A}_1$ consists of "propositions" of three kinds. A "v-proposition" specifies the value of a fluent in the initial situation, or after performing a sequence of actions. An "ef-proposition" describes the effect of an action on a fluent. A "ex-proposition" describes when an action is executable.

We begin with two disjoint nonempty sets of symbols, called *fluent names* and *action names*. A *fluent literal* is either a fluent name or a fluent name preceded by $\neg$. For a fluent literal l, by $\bar{l}$ we denote the fluent literal $\neg l$, and we shorten $\neg\neg l$ to l.

A *v-proposition* is an expression of the form $\quad f$ **after** $a_1, \ldots, a_m, \qquad (1)$
where f is a fluent literal, and $a_1,\ldots,a_m$ $(m \geq 0)$ are action names.
If $m = 0$, we will write (1) as **initially** f.

An *ef-proposition* is an expression of the form $\quad a$ **causes** f **if** $p_1, \ldots, p_n, \qquad (2)$
where a is an action name, and each of $f, p_1, \ldots, p_n$ $(n \geq 0)$ is a fluent literal. About this proposition we say that it *describes the effect of a on f*, and that $p_1, \ldots, p_n$ are its *preconditions*. If $n = 0$, we will drop **if** and write simply a **causes** f.

relate planning algorithms to resolution strategies. We believe our work is complementary to this work.

[4]Note that unlike logic programs, PROLOG is a programming language with many commercially available efficient interpreters and compilers.

Two ef-propositions with preconditions $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ respectively are said to be *contradictory* if they describe the effect of the same action a on complementary fs, and $\{p_1, \ldots, p_n\} \cap \{\overline{q_1}, \ldots, \overline{q_m}\} = \emptyset$

A *ex-proposition* is an expression of the form **executable** a **if** $q_1, \ldots q_n$, (3)
where a is an action name, and each of $q_1, \ldots, q_n$ $(n \geq 0)$ is a fluent literal. About this proposition we say that it *stipulates that a is executable only in situations where $q_1, \ldots, q_n$ are satisfied.*

A *proposition* is a v-proposition, ex-proposition, or an ef-proposition. A *domain description*, or simply *domain*, is a set of propositions which does not contain contradictory ef-propositions, and which contains at least one ex-proposition for each action name.

Domain descriptions in which each v-proposition has the form **initially** f correspond to temporal projection problems, and we will call them *projection domains*. Domain descriptions that contain complete information about the initial state (i.e. for any fluent name f, either **initially** f or **initially** $\neg f$ is in the domain description, but not both) are referred to as *strongly complete*.

2.2 Semantics

A *state* is a set of fluent names. Given a fluent name f and a state σ, we say that f *holds* in σ if $f \in \sigma$; $\neg f$ *holds* in σ if $f \notin \sigma$. A *transition function* is a mapping Φ of the set of pairs (a, σ), where a is an action name and σ is a state, into the set of states. A *structure* is a pair (σ_0, Φ), where σ_0 is a state (the *initial state* of the structure), and Φ is a transition function.

We say that a sequence of actions $a_1, \ldots, a_m$ is executable in a structure $M = (\sigma_0, \Phi)$ if for every $1 \leq k \leq m$, $\Phi(a_k, \Phi(a_{k-1}, \ldots, \Phi(a_1, \sigma_0) \ldots))$, is defined. The corresponding state is denoted by $M^{(a_1, \ldots, a_k)}$. We say that a v-proposition (1) is *true* (*false*) in a structure M if $a_1, \ldots, a_m$ is executable in M and f holds in the state $M^{(a_1, \ldots, a_m)}$. In particular, the proposition **initially** f is true in M iff f holds in the initial state of M.

A structure (σ_0, Φ) is a *model* of a domain description D if the following two conditions are satisfied: (i) every v-proposition from D is true in (σ_0, Φ); and
(ii) for every action a and every state σ,
(a) if the preconditions of the ex-proposition corresponding to a hold in σ then $\Phi(a, \sigma) = \sigma \cup \sigma' \setminus \sigma''$,
where σ' (σ'') is the set of fluent names f such that D includes an ef-proposition describing the effect of a on f (respectively, $\neg f$) whose preconditions hold in σ; else,
(b) $\Phi(a, \sigma)$ is undefined.

The condition (ii) characterizes Φ uniquely. Consequently, different models of the same domain description can differ only by their initial states. A domain description is *consistent* if it has a model, and *complete* if it has exactly one model.

A v-proposition is *entailed* by a domain description D if it is true in every model of D.

Proposition 2.1 A strongly complete and consistent domain description has exactly one model. $\square$

Notice that so far we considered fluents and actions to be propositional. We extend our language to allow variables by considering them as schema variables. (This is similar to the approach in [GL88].) The semantics of domain descriptions in this extended language is then defined with respect to the instantiated domain descriptions. (Note that the instantiated domain description may then be infinite.)

2.3 Describing Actions by Logic Programs : the translation π

In this section we describe a translation of domain descriptions in $\mathcal{A}_1$ to logic programs. The logic program πD, corresponding to a domain description D, uses variables of three sorts: *situation* variables $S, S', \ldots$, *fluent* variables $F, F', \ldots$, and *action* variables $A, A', \ldots$. It includes the situation constant s_0, and the fluent names and action names of D, that become object constants of the corresponding sorts. The predicate *holds* has two arguments; the first argument is a fluent literal, and the second is a situation. For example, we can have $holds(\neg alive, s_0)$, where *alive* is a fluent, and $\neg alive$ is a negative fluent literal. There are also some other predicate and function symbols; the sorts of their arguments and functions will be clear from their use in the rules below.

The program πD will consist of the translations of the individual propositions from D and the standard rule of inertia. The following notations will be useful: If $a_1, \ldots, a_m$ are action names, $[a_1, \ldots, a_m]$ stands for the ground term $result(a_m, result(a_{m-1}, \ldots, result(a_1, s_0)\ldots))$.

For any domain description D, the translation πD consists of the following rules:

1. An inertia rule of the form:

$holds(F, result(A, S)) \leftarrow holds(F, S), reachable(res(A, S)), not\ \ ab(F, A, S)$

2. For each v-proposition of the form (1), if $m = 0$ then πD contains the rule

$holds(f, s_0) \leftarrow$

otherwise πD contains the rules

$holds(f, [a_1, \ldots, a_m]) \leftarrow reachable([a_1, a_2, \ldots, a_m])$

$ab(\overline{f}, a_m, [a_1, \ldots, a_{m-1}]) \leftarrow$

3. For each ef-proposition (2) πD contains the rules:

$holds(f, result(a, S)) \leftarrow reachable(res(a, S)), holds(p_1, S), \ldots, holds(p_n, S)$

$ab(\overline{f}, a, S) \leftarrow not\ \ holds(\overline{p_1}, S), \ldots, not\ \ holds(\overline{p_n}, S)$

4. For each ex-proposition of the form (3) πD contains the rule

$reachable(res(a, S)) \leftarrow holds(q_1, s), \ldots, holds(q_n, s), reachable(S)$

5. Finally, we also have the rule $\qquad reachable(s_0) \leftarrow$

Proposition 2.2 πD is an *acyclic* [AB90] logic program. □

Acyclicity guarantees that SLDNF resolution is sound and complete with respect to πD with respect to the stable model semantics. (In fact almost all major semantics such as completion semantics, stable model semantics, well-founded semantics etc. coincide for acyclic programs.) Acyclicity [AB90] guarantees decidability for queries (using SLDNF resolution) that do not flounder. Hence, acyclicity of πD guarantees that plan existence is decidable. (This approach can also be used to extend decidability results in [ENS95] to extended languages, such as when restricted ramification rules are added.)

Moreover, for a query Q of the form $\leftarrow holds(f, [a_1, \ldots, a_n])$ the instantiated rules in πD that are necessary to evaluate Q are polynomial in size with respect to the size of πD (not with respect to the instantiated πD, which is infinite in size) and Q. This is evident from the fact that the variable S in all the rules is πD need only be instantiated with n different values; the n prefixes of $[a_1, \ldots, a_n]$. Since, acyclicity implies that each instantiated rule be used only once (when using SLDNF with tabling [CSW95]) answering Q (i.e. testing) is polynomial time[5].

Proposition 2.3 For consistent domain descriptions D,
(i) πD is consistent with respect to holds i.e. it is not the case that for any f and any s, $\pi D \models holds(f, S)$ and $\pi D \models holds(\overline{f}, S)$.
(ii) πD is sound. i.e. If $\pi D \models holds(f, [a_1, \ldots, a_n])$ then $D \models f$ **after** $a_1, \ldots, a_n$. □

Proposition 2.4 For strongly complete and consistent domains D, πD is sound and complete. i.e. $\pi D \models holds(f, [a_1, \ldots, a_n])$ iff $D \models f$ **after** $a_1, \ldots, a_n$. □

Proof (sketch): The proof of the above two propositions is done by induction on the size of the action sequences, and extensively uses a lemma from [MS89] which relates the presence of atoms in the head and body of a rule in the program to their presence in a stable model of the program. □

Corollary 2.1 For strongly complete and consistent domains D, πD is equivalent to $\pi' D$, which is same as πD except that *not* $holds(\overline{p_i}, S)$ is replaced by $holds(p_i, S)$ (for $1 \leq i \leq n$) in the body of the rules in item (3) where ab is in the head. □

2.4 Simpler translations for particular subclasses

In this section we consider several simple classes of projection domains and give sound and complete translations to logic programs. These translations are simpler than π.

We assume that domain descriptions are not only projection domains but also are strongly complete. With these assumption we identify four domains, *Vanilla, $\mathcal{A}$[6], STRIPS and CSTRIPS* domains based on whether conditional effects are present (i.e. the ef-propositions

[5]Note that we are referring to expression complexity rather than data complexity [AHV95]. In our case the program is given and the size of the query varies while in databases, since function symbols are not allowed the size of the query is fixed while the size of the data varies.

[6]We call them $\mathcal{A}$ domains because of its close similarity with the language $\mathcal{A}$ [GL93] where actions are always assumed to be executable.

have preconditions or not) and if the actions have executability conditions (i.e. the ex-propositions have preconditions or not). In the Vanilla domain ef-propositions do not have preconditions and actions are always executable. In the STRIPS domain the ef-propositions have empty preconditions (i.e. no conditional effects) but actions do have executability conditions. In the $\mathcal{A}$ domain actions do not have executability conditions but do have conditional effects and in CSTRIPS actions have both conditional effects and executability conditions.

$\pi_v : for$ **Vanilla** $domains - empty$ $preconditions\ in\ ef\ and\ ex - propositions$	$\pi_a : for\ \mathcal{A} -\ domains\ empty\ preconditions\ in$ $ex - propositions$
$\underline{Inertia}$ $r_1 :\ holds(F, res(A,S)) \leftarrow holds(F,S),$ $not\ ab(F,A)$ $\underline{Translating\ v - propositions}$ $r_2 :\ holds(f, s_0) \leftarrow\quad if$ **initially** $f \in D$ $\underline{Translating\ ef - propositions}$ $For\ every\ action\ a\ with\ effect\ f$ $r_3 :\ holds(f, res(a,S)) \leftarrow$ $r_3' :\ ab(\overline{f}, a) \leftarrow$	$\underline{Inertia}$ $r_1 :\ holds(F, res(A,S)) \leftarrow holds(F,S),$ $not\ ab(F,A,S)$ $\underline{Translating\ v - propositions}$ $r_2 :\ holds(f, s_0) \leftarrow\quad if$ **initially** $f \in D$ $\underline{Translating\ ef - propositions}$ $For\ every\ action\ a\ with\ conditional\ effect\ f$ $and\ with\ conditions\ p_1, \ldots, p_n$ $r_3 :\ holds(f, res(a,S)) \leftarrow holds(p_1,S), \ldots,$ $holds(p_n, S)$ $r_3' :\ ab(\overline{f}, a, S) \leftarrow holds(p_1,S), \ldots, holds(p_n,S)$
$\pi_s : for$ **STRIPS** $domain\ - empty$ $preconditions\ in\ ef - propositions$	$\pi_c : for$ **CSTRIPS** $domains$
$\underline{Inertia}$ $r_1 :\ holds(F, res(A,S)) \leftarrow holds(F,S),$ $reachable(res(A,S)),$ $not\ ab(F,A)$ $\underline{Translating\ v - propositions}$ $r_2 :\ holds(f, s_0) \leftarrow\quad if$ **initially** $f \in D$ $\underline{Translating\ ef - propositions}$ $For\ every\ action\ a\ with\ effect\ f$ $r_3 :\ holds(f, res(a,S)) \leftarrow reachable(res(a,S))$ $r_3' :\ ab(\overline{f}, a) \leftarrow$ $\underline{Translating\ ex - propositions}$ $For\ every\ action\ a\ with$ $preconditions\ q_1, \ldots q_n$ $r_4 :\ reachable(res(a,S)) \leftarrow holds(q_1,S), \ldots,$ $holds(q_n,S), reachable(S)$ $r_5 :\ reachable(s_0) \leftarrow$	$\underline{Inertia}$ $r_1 :\ holds(F, res(A,S)) \leftarrow holds(F,S),$ $reachable(res(A,S)),$ $not\ ab(F,A,S)$ $\underline{Translating\ v - propositions}$ $r_2 :\ holds(f, s_0) \leftarrow\quad if$ **initially** $f \in D$ $\underline{Translating\ ef - propositions}$ $For\ every\ action\ a\ with\ conditional\ effect\ f$ $with\ conditions\ p_1, \ldots, p_n$ $r_3 :\ holds(f, res(a,S)) \leftarrow reachable(res(a,S)),$ $holds(p_1,S), \ldots, holds(p_n,S)$ $r_3' :\ ab(\overline{f}, a, S) \leftarrow holds(p_1,S), \ldots, holds(p_n,S)$ $\underline{Translating\ ex - propositions}$ $For\ every\ action\ a\ with$ $preconditions\ q_1, \ldots q_n$ $r_4 :\ reachable(res(a,S)) \leftarrow holds(q_1,S), \ldots,$ $holds(q_n,S), reachable(S)$ $r_5 :\ reachable(s_0) \leftarrow$

The above table shows the translations π_v, π_s, π_a, and π_c for translating vanilla, STRIPS, $\mathcal{A}$ and CSTRIPS domains.

Proposition 2.5 The translations π_v, π_s, π_a and π_c are sound and complete with respect to the entailment $\models$ for domain descriptions in appropriate domains (i.e. in domains vanilla, STRIPS, $\mathcal{A}$ and CSTRIPS respectively). $\Box$

3 Partial Order Planning

In this section we briefly describe partial order planing algorithms for STRIPS and for STRIPS extended with conditional effects. Due to lack of space our exposition which is based on [Wel94] is short and only meant to be a quick reference for readers already familiar with similar algorithms.

In later sections we refer to the algorithms in this section to show how they can be used for plan testing, and how and which steps in the algorithms need to be extended so as to be able to construct plans when we have extended ontologies.

3.1 Without Conditional Effects (STRIPS)

Algorithm 3.1 $POP(< A, O, L >, Agenda, AList)$

A: A set of instantiated actions;

O: A set of orderings of the form $a_1 < a_2$ between instantiated actions where $a_1 < a_2$ means a_1 occurs before a_2.

L: A set of causal links of the forms $a_1 \xrightarrow{f} a_2$ which means a_1 should occur before a_2 to satisfy one of the preconditions f of a_2.

Agenda: A collection of pairs $< Q, a >$, where $a \in A$, and Q is a precondition of a. Intuitively it means that the algorithm must try to achieve Q so that a can be executed.

Alist: A list of actions.

Step 1: (Termination) If Agenda is empty then return $< A, O, L >$.

Step 2: (Goal Selection) Select a pair $< Q, a >$ from the agenda. (Not a backtracking point.)

Step 3: (Action Selection) Choose an action a_{add} that adds Q either by instantiating an action in *Alist* or from A which can be consistently ordered before a.
If no such action exists, then return failure.
Otherwise, $L' = L \cup \{a_{add} \xrightarrow{Q} a\}$, $O' = O \cup \{a_{add} < a, a_0 < a_{add}, a_{add} < a_\infty\}$, and $A' = A \cup \{a_{add}\}$.

Step 4: (Updating Goal state) $Agenda' = Agenda \setminus \{< Q, a >\}$
If a_{add} is newly instantiated, then for each conjunct Q_i of its precondition add $< Q_i, a_{add} >$ to $Agenda'$.

Step 5: (Causal Link Protection) For every action a_t that might threaten a causal link $a_p \xrightarrow{R} a_c$, add a consistent ordering constraint, either
(a) Demotion: Add $a_t < a_p$ to O' or (b) Promotion: Add $a_c < a_t$ to O'.
If neither constraint in consistent, then return failure.

Step 6: (Recursive invocation) $POP(< A', O', L' >, Agenda', Alist)$. $\Box$

3.2 With Conditional Effects

In the presence of conditional effects, the modified algorithm is as follows:

Algorithm 3.2 $POP_c(< A, O, L >, agenda, AList)$

Step 1 and **Step 2** are as in Algorithm 3.1.

Step 3: (Action Selection) Choose an action a_{add} that adds or *(conditionally) adds* Q either by instantiating an action in *Alist* or from A which can be consistently ordered before a. If no such action exists, then return failure. Otherwise, $L' = L \cup \{a_{add} \xrightarrow{Q} a\}$, $O' = O \cup \{a_{add} < a, a_0 < a_{add}, a_{add} < a_\infty\}$, and $A' = A \cup \{a_{add}\}$.

Step 4: (Updating Goal state) $Agenda' = Agenda \setminus \{< Q, a >\}$
If Q is a conditional effect and it has not been already used to establish a link of the form $a_{add} \xrightarrow{Q} A$, for some A, then for each conjunct Q_i in the condition add $< Q_i, a_{add} >$ to $Agenda'$. If a_{add} is newly instantiated, then for each conjunct Q_i of its precondition add $< Q_i, a_{add} >$ to $Agenda'$.

Step 5: (Causal Link Protection) For every action a_t that might threaten a causal link $a_p \xrightarrow{R} a_c$, add one of the following consistent constraint
(a) Demotion: Add $a_t < a_p$ to O' or (b) Promotion: Add $a_c < a_t$ to O', or.
(c) Confrontation: If a_t's threatening effect is conditional with antecedent S, then non-deterministically choose some conjunct g from S and add $(\neg g, a_t)$ to $Agenda'$.
If none of the constraint is consistent, then return failure.

Step 6: (Recursive invocation) $POP_c(< A', O', L' >, Agenda', Alist)$. $\Box$

4 Relating logic programming theories of actions and partial oder planners

In general theories of actions provide us with an entailment relation using which we can entail whether a formula is entailed by a theory or not. When the formula is of the form $\exists X holds(f, X)$ then its entailment corresponds to plan existence, and when the formula is of the form $holds(f, [a_1, \ldots, a_n])$ then its entailment corresponds to testing. The general purpose procedural methods (such as resolution for theories in first-order logic and SLDNF for theories in logic programming) corresponding to the entailment relations are very inefficient when used for query answering (i.e., in our case finding a plan), and that was the reason other methods to find plan were explored in the planning community which gave rise to many planning algorithms. But as we show in Section 2, for certain classes of theories we can efficiently do testing.

Therefore, the approach we take in extending planning algorithms to larger languages is to investigate the correspondence between testing with respect to the logic programs (obtained by the translations in Section 2.4) and testing using partial order planners. This is the goal of this section.

Although partial order planners do not do any direct testing, it is important to discuss how they can be used for testing. This will enable us to answer questions such as, "where is the frame problem in partial order planners"[7], and also will allow us to modify existing partial order planners to allow extended languages.

Testing with respect to the logic programming theories of actions corresponds to finding if the logic program entails $holds(f, [a_1, \ldots, a_n])$. Now the question is how do we reason about the truth of $holds(f, [a_1, \ldots, a_n])$ in a POP.

When dealing with vanilla domains (i.e. when actions do not have conditional effects or preconditions) the following call to POP will determine if $holds(f, [a_1, \ldots, a_n])$ is true.

$POP(< \{a_0, a_1, \ldots, a_n, a_\infty\}, \{a_0 < a_1, a_1 < a_2, \ldots, a_n < a_\infty\}, \emptyset\} >, \{(f, a_\infty)\}, Alist)$

We make a restriction that we do not choose new instantiations of actions in Step 3 and for statements of the form **initially** h in our theory we include h as an effect of a_0.

But when actions have preconditions the above call to POP is not sufficient as its Agenda does not include the precondition of the various actions. Instead the following call with $Agenda^*$ defined as the set $\{< Q, a_i >: \ 1 \leq i \leq n$ and Q is a precondition of $a_i\}$ will be able to determine if $holds(f, [a_1, \ldots, a_n])$ is true or not.

$POP(< \{a_0, a_1, \ldots, a_n, a_\infty\}, \{a_0 < a_1, a_1 < a_2, \ldots, a_n < a_\infty\}, \emptyset\} > Agenda^* \cup \{(f, a_\infty)\}, Alist)$

We are now ready to show the correspondence between the execution following the above mentioned calls to POP and the execution of the query $\leftarrow holds(f, [a_1, \ldots, a_n])$ with respect to the appropriate logic programs.

4.1 Testing in STRIPS domains

In this section we consider planning problems which can be described in the STRIPS domain. For such a domain description D, $\pi_s D$ is an acyclic program and hence SLDNF resolution is sound and complete with respect to it. Moreover, testing is polynomial with respect to $\pi_s D$.

To find out if $holds(f, [a_1, \ldots a_n])$ is true in the logic program we now need to find an a_p such that

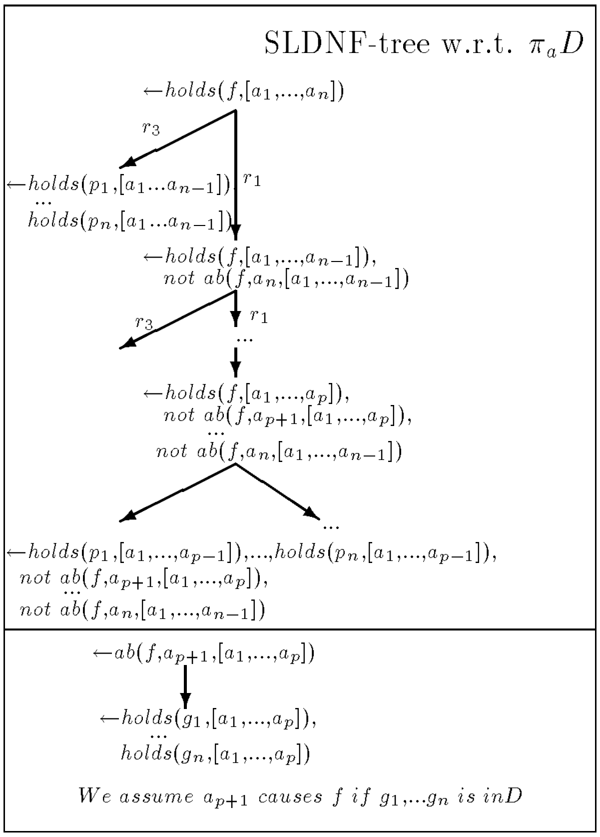
(i) f is an effect of a_p (which means $holds(f, [a_1, \ldots, a_p])$ is true), such that all of $ab(f, a_{p+1}), \ldots, ab(f, a_n)$ are false, (This would ensure that by repeatedly using the inertia rules we will obtain $holds(f, [a_1, \ldots a_n])$ to be true. If we do not find such an a_p then we will conclude $holds(f, [a_1, \ldots a_n])$ to be $false$.), and for all i, $p < i \leq n$ $reachable([a_1, \ldots, a_i])$ is true.

Now let us consider what the call

$POP(< \{a_0, a_1, \ldots, a_n, a_\infty\}, \{a_0 < a_1, a_1 < a_2, \ldots, a_n < a_\infty\}, \emptyset > Agenda^* \cup \{(f, a_\infty)\}, Alist)$

with $Agenda^*$ defined as the set $\{< Q, a_i >: \ 1 \leq i \leq n$ and Q is a precondition of $a_i\}$

[7] A partial answer to this is suggested by the modal truth criteria [KN94, Cha87].



does to determine the truth of $holds(f, [a_1, \ldots a_n])$.

In Step 3 of the algorithm it chooses an a_p such that a_p adds f. It then needs to check that none of the actions in $\{a_{p+1}, \ldots, a_n\}$ threatens the link $a_p \xrightarrow{f} a_\infty$.

• *Checking if* a_{p+1} *threatens the link* $a_p \xrightarrow{f} a_\infty$ *is equivalent to determining if* $ab(f, a_{p+1})$ *is true in the logic program* $\pi_s D$*.*

• To check that for all i, $p < i \leq n$ $reachable([a_1, \ldots, a_i])$ is true, the logic program has to check that for each of these i's, $holds(Q, [a_1, \ldots, a_{i-1}])$ is true where Q is a condition for the executability of a_i. This corresponds to the pair $< Q, a_i >$ in the $Agenda^*$.

The above two bulleted items correspond to the bottom leaf node of the SLDNF tree with respect to $\pi_s D$ in the following figure (in the next page).

4.2 Testing in $\mathcal{A}$ domains

In this section we consider planning problems which can be described in the $\mathcal{A}$ domain. For such a domain description D, $\pi_a D$ is an acyclic program and hence SLDNF resolution is sound and complete with respect to it. Moreover, testing is polynomial with respect to $\pi_a D$.

To find out if $holds(f, [a_1, \ldots a_n])$ is true in the logic program we now need to find an a_p such that f is an effect of a_p (which means $holds(f, [a_1, \ldots, a_p])$ is true), and such that all of $ab(f, a_{p+1}, [a_1, \ldots, a_p]), \ldots, ab(f, a_n, [a_1, \ldots, a_{n-1}])$ are false. (This would ensure that by repeatedly using the inertia rules we will obtain $holds(f,[a_1,\ldots a_n])$ to be true. If we do not find such an a_p then we will conclude $holds(f,[a_1,\ldots a_n])$ to be $false$.)

Now let us consider what the call

$POP(<\{a_0,a_1,\ldots,a_n,a_\infty\},\{a_0<a_1,a_1<a_2,\ldots,a_n<a_\infty\},\emptyset\}>,\{(f,a_\infty\},Alist)$

does to determine the truth of $holds(f,[a_1,\ldots a_n])$.

In Step 3 of the algorithm it chooses an a_p such that a_p conditionally adds f. For each of the condition Q it adds the pair $<Q,a_p>$ to the Agenda'. It then needs to check that none of the actions in $\{a_{p+1},\ldots,a_n\}$ threatens the link $a_p \xrightarrow{f} a_\infty$.

• *Checking if* a_{p+1} *might threaten the link* $a_p \xrightarrow{f} a_\infty$ *is equivalent to finding a rule whose head is* $ab(f,a_{p+1},[a_1,\ldots,a_p])$ *in the logic program* $\pi_a D$. This is evident the bottom leaf node of the SLDNF tree with respect to $\pi_a D$.

Since the ordering between the action steps rules out promotion and demotion, confrontation is the only option to protect the causal link. As a result POP chooses a condition g_i of the effect f for the action a_{p+1} and adds $(\neg g_i, a_{p+1})$ to the Agenda.

• *Selecting* $(\neg g_i,a_{p+1})$ *in Step 2 in a later execution of the algorithm is equivalent to proving* $\underline{holds(g_i,[a_1,\ldots,a_p])\ is\ false}$ *which will guarantee that* $ab(f,a_{p+1},[a_1,\ldots,a_p])$ *is* $false$*. This is evident from the small box in the figure.*

5 POP with Incomplete Initial State

Let us now remove the strong completeness assumption (i.e. the initial state may be incomplete[8], but stay with consistent projection domains. From Proposition 2.3 πD will be sound (but no longer complete).

When we put additional restrictions about the ef and ex-propositions similar to the ones put in the vanilla, STRIPS, $\mathcal{A}$ and CSTRIPS domains, the only change we need is in π_a (no changes are necessary in π_v and π_s), where we change the body of the rule with ab in the head by replacing $holds(p_i,s)$ by *not* $holds(\overline{p_i},s)$ for $1 \leq i \leq n$. Lets call this translation as π'_a. (Gelfond and Lifschitz [GL93] noticed the impact of this change and used *not* $holds(\overline{p_i},s)$ in their logic programming formalization of $\mathcal{A}$. They explain in details why using *not* $holds(\overline{p_i},s)$ is the correct approach.)

Consider the underlined portion of the second bulleted item in Section 4.2. While with respect to $\pi_a D$ we were proving $\underline{holds(g,[a_1,\ldots,a_p])\text{ to be }false}$, with respect to $\pi'_a D$ we need to prove $\underline{holds(\overline{g},[a_1,\ldots,a_p])\text{ is true}}$. This directly corresponds to selecting $(\neg g,a_{p+1})$ in Step 2 of POP_c.

This suggests that POP and POP_c can be used to find plans even in the absence of complete information (Recall the last footnote which states the restricted kind of incompleteness we

[8]Note that we are referring to a particular kind of incompleteness [GL91] where we know that some fluents are true in the initial situation, some fluents are false and the truth value of the rest of the fluents is unknown. This is a special kind of incompleteness which is more tractable than when we have incompleteness due to disjunctive information about the initial state.

have in mind.) about the initial state. The algorithms will be sound in the sense that if they find a plan the plan would be correct. But they will not necessarily always find a plan.

6 Using LP theories of Actions to extend POP

We are now ready to discuss how logic programming theories of actions can be used to extend POP so as to be able to plan in an extended domain.

6.1 Disjunctive preconditions and conditional effects

Notice that in none of the domains that we discussed (vanilla, STRIPS, $\mathcal{A}$ and CSTRIPS) we required that there can be only one ef-proposition for each pair of action and fluent. We also did not require that there be only one ex-proposition with respect to each action. This means that if we have two ex-propositions

executable a **if** $q_1, \ldots, q_n$ and **executable** a **if** $r_1, \ldots, r_p$,

then in effect we are saying that a is executable if $(q_1 \wedge \ldots \wedge q_n) \vee (r_1 \wedge \ldots \wedge r_n)$ is true.

This suggests that we can make minor changes to our POP and POP_c to accept such an extended domain. Moreover, the minor changes can be obtained by looking at the SLDNF tree of the corresponding logic program.

In the above example the translation $\pi_s D$ will have two rules with $reachable(res(a, S))$ in the head. Hence, during the SLDNF resolution there will be an or-branching. This corresponds to modifying Step 4 of the POP algorithm where we can non-deterministically choose to either add $\{< q_1, a_{add} >, \ldots, < q_n, a_{add} >\}$ or add $\{< r_1, a_{add} >, \ldots, < r_p, a_{add} >\}$ to the $Agenda'$

Similar changes can be made in POP_c to take care of disjunctive preconditions, and conditional effects where the conditions may be disjunctive.

The above extension of POP is taken care of in UCPOP. Let us discuss an extension that is not taken care of in UCPOP.

6.2 Using learned hypothetical facts

Recall that so far we are only concentrating on projection domains, i.e. we have been only allowing v-propositions about the initial state. Suppose we would like to remove this restriction. But then what will a v-proposition f **after** $a_1, \ldots, a_n$ correspond to?

It corresponds to the fact that we are told or somehow the planner learnt that executing $a_1 \ldots a_n$ sequentially from the initial state will make f true. Our goal now is to extend POP and POP_c to be able to use such facts.

When we use such facts in πD the logic program is only sound (not complete). This means our extension of POP and POP_c suggested πD will only be sound.

The extension to POP suggested by πD is to expand step 3 where not only we can choose an action but can also nondeterministically choose a sequence of actions. The updating of L', O', A' and $Agenda'$ in steps 3 and 4 is modified appropriately.

6.3 Acyclic ramifications

Let us now extend our language to allow a particular restricted kind of ramifications (Unrestricted ramifications are discussed in [KL94, Bar95]). We will now have two kind of fluents: *basic fluents* and *derived fluents*. The fluent f in the v-propositions (1) and the ef-propositions (2) will be required to be basic fluents, while the fluents p_i's and q_i's in the ef- and ex-propositions can be either basic or derived fluents.

Besides the v-, ef- and ex-propositions we allow a new kind of proposition called c-propositions (where 'c' stands for constraint) of the form p **if** $p_1, \ldots, p_k$, (4)
where p is a derived fluent and $p_1, \ldots, p_n$ are either basic fluents or derived fluents. We will require that the domain descriptions do not have contradictory (similar to the definition in Section 2.1) c-propositions.

The set of c-propositions in a domain description will be said to be acyclic if we can assign each fluent to a natural number (called the level) such that for any c-proposition the level of the fluent in the head is higher than the level of any of the fluents in the body. This will guarantee that when we translate a domain description to a logic program with the added changes that c-propositions of the form (4) are translated to the rule

$holds(p, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S)$

and inertia rule is only for the basic fluents. The resultant program will be still acyclic and hence will retain the decidability property. The program will also retain the polynomial time testing property.

To incorporate this extension to the POP, step 3 of the algorithm will be modified where if the Q selected in step 2 is a derived fluent, then if there is a c-constraint with Q in the head and the fluents $p_1, \ldots, p_n$ in the body then $< p_1, a >, \ldots, < p_n, a >$ are added to the Agenda. The rest of step 3 and the whole algorithm remains unchanged.

7 Conclusion

In this paper we extended the language $\mathcal{A}$ to include executability conditions and provided translation to a logic program. We provided several simple translation corresponding to particular restricted domains. We showed the correspondence between doing projection with respect to the logic programs and with respect to the planners. We discussed how these correspondence can be used to extend planning methodologies to larger domains. We briefly discussed three such extensions: having disjunction in the preconditions and conditional effects with disjunctive conditions; having facts about the effect of sequence of actions; and having restricted ramifications.

Our next goal is to consider concurrent actions and use the logic programming theory in [BG93] to extend partial order planners to be able to plan in presence of concurrent actions.

This is necessary in many cases when actions needed to be executed concurrently to achieve certain effects. For example, consider two actions called $left\_lift$ and $right\_lift$ which individually cause a table to be titled while executed together they cause the table to be lifted. If our goal is to lift the table we must construct a plan which does $left\_lift$ and $right\_lift$ concurrently, not in any sequential order.

In summary, we would like to advocate the following methodology. While formulating new theories in reasoning about actions subclasses need to be identified that guarantee decidability of the plan-existence problem and polynomial time testing. Furthermore we believe that logic programming theories of actions makes this identification easier. (Note that the decidability proofs in [ENS95] critically depend on properties of logic programs.) Also, in many cases the SLDNF tree helps us to identify the modification necessary to the original POPs so as to accept an extended language.

References

[AB90] K. Apt and M. Bezem. Acyclic programs. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 617–633, 1990.

[AB91] K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.

[AHV95] S. Abiteboul, R. Hall, and V. Vianu, editors. *Foundations of Databases*. Addison Wesley, 1995.

[AP94] K. Apt and A. Pellegrini. On the occur-check free logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):687–726, 1994.

[Bar95] C. Baral. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification. In *IJCAI 95*, pages 2017–2023, 95.

[BF95] A. Blum and M. Furst. Fast planning through planning graph analysis. In *IJCAI 95*, pages 1636–1642, 95.

[BG93] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proc. of 13th International Joint Conference on Artificial Intelligence, Chambery, France*, pages 866–871, 1993.

[BG94] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[BGP95] C. Baral, M. Gelfond, and A. Provetti. Representing Actions I: Laws, Observations and Hypothesis. In *Proc. of AAAI 95 Spring Symposium on Extending Theories of Action: Formal theory and practical applications*, http://cs.utep.edu/chitta/publications.html, 1995.

[Bro87] F. Brown, editor. *Proceedings of the 1987 worskshop on The Frame Problem in AI*. Morgan Kaufmann, CA, USA, 1987.

[Cha87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, pages 333–377, 1987.

[CSW95] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[DDS93] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proceedings of ILPS 93, Vancouver*, pages 147–164, 1993.

[ENS92] K. Erol, D. Nau, and V.S. Subrahmanian. On the complexity of domain-independent planning. In *AAAI 92*, pages 381–386, 92.

[ENS95] K. Erol, D. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *To appear in AI Journal (Also Univ of Maryland CS TR-2797)*, 95.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387, 1991.

[GL93] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

[GL95] E. Giunchiglia and V. Lifschitz. Dependent fluents. In *IJCAI 95*, pages 1964–1969, 95.

[HT93] S Holldobler and M Thielscher. Actions and specificity. In D. Miller, editor, *Proc. of ICLP-93*, pages 164–180, 1993.

[KKY95] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *AI Journal (to appear), also in ASU-CSE-TR 94 002*, 1995.

[KL94] G. Kartha and V. Lifschitz. Actions with indirect effects: Preliminary report. In *KR 94*, pages 341–350, 1994.

[KN94] S. Kambhampati and D. Nau. On the nature of modal truth criteria in planning. In *Proc. of AAAI-94*, pages 1055–1060, 1994.

[Lin95] F. Lin. An ordering on goals for planning - formalizing control information in the situation calculus, 1995. manuscript.

[Lin95] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI 95*, pages 1985–1993, 95.

[LLL+94] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high level robot programming – a progress report. In *Working notes of the 1994 AAAI fall symposium on Control of the Physical World by Intelligent Systems (to appear), New Orleans, LA*, November 1994.

[LR94] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, october 1994.

[LS92] F. Lin and Y. Shoham. Concurrent actions in the situation calculus. In *Proc. of AAAI–92*, pages 590–595, 1992.

[MS89] W. Marek and V.S. Subrahmanian. The relationship between logic program semantics and non–monotonic reasoning. In G. Levi and M. Martelli, editors, *Proc. of the Sixth Int'l Conf. on Logic Programming*, pages 600–617, 1989.

[MS94] R. Miller and M. Shanahan. Narratives in the situation calculus. *Journal of Logic and Computation*, 4(5):513–530, october 1994.

[MT95] N. McCain and M. Turner. A causal theory of ramifications and qualifications. In *IJCAI 95*, pages 1978–1984, 95.

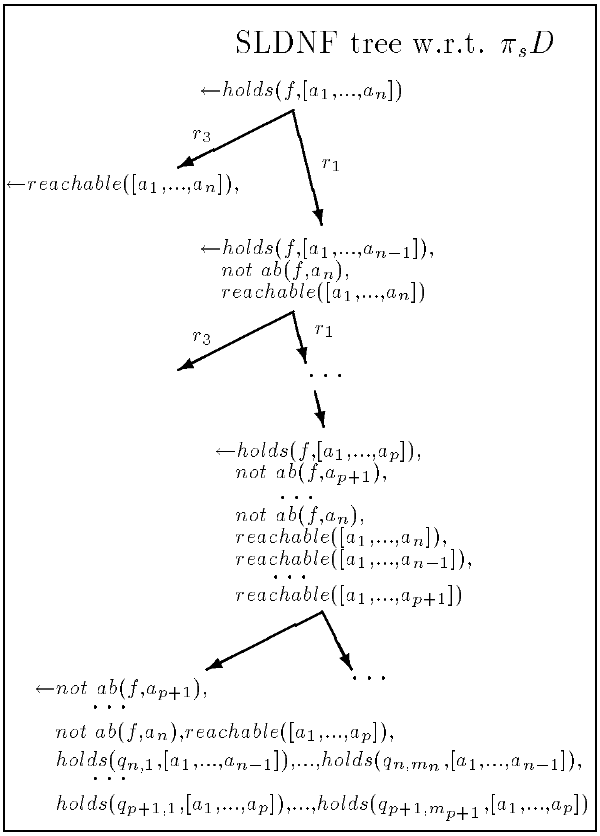
[Ped94] E. Pednault. ADL and the State-transition model of actions. *Journal of logic and computation*, 4(5):467–513, october 1994.

[Pin94] J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Department of Computer Science, February 1994. KRR-TR-94-1.

[PR93] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Proceedings of 10th International Conference in Logic Programming, Hungary*, pages 203–221, 1993.

[SL93] R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *AAAI 93*, pages 689–695, 1993.

[Wel94] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, winter 1994.


SLDNF tree w.r.t. $\pi_s D$
$\leftarrow holds(f,[a_1,...,a_n])$
r_3
r_1
$\leftarrow reachable([a_1,...,a_n]),$
$\leftarrow holds(f,[a_1,...,a_{n-1}]),$
$not\ ab(f,a_n),$
$reachable([a_1,...,a_n])$
r_3
r_1
$\ldots$
$\leftarrow holds(f,[a_1,...,a_p]),$
$not\ ab(f,a_{p+1}),$
$\ldots$
$not\ ab(f,a_n),$
$reachable([a_1,...,a_n]),$
$reachable([a_1,...,a_{n-1}]),$
$\ldots$
$reachable([a_1,...,a_{p+1}])$
$\ldots$
$\leftarrow not\ ab(f,a_{p+1}),$
$\ldots$
$not\ ab(f,a_n), reachable([a_1,...,a_p]),$
$holds(q_{n,1},[a_1,...,a_{n-1}]),...,holds(q_{n,m_n},[a_1,...,a_{n-1}]),$
$\ldots$
$holds(q_{p+1,1},[a_1,...,a_p]),...,holds(q_{p+1,m_{p+1}},[a_1,...,a_p])$