Baltzer Journals AMAI, Vol. 19, No. 3-4, 1997

The Logic of Totally and Partially Ordered Plans: A Deductive Database Approach

ANTONIO BROGI [1] V.S. SUBRAHMANIAN [2] AND CARLO ZANIOLO [3]

[1] *Dipartimento di Informatica, Università di Pisa*
Corso Italia 40, 56125 Pisa, Italy
E-mail: brogi@di.unipi.it

[2] *Computer Science Department, University of Maryland*
College Park, MD 20742, U.S.A.
E-mail: vs@cs.umd.edu

[3] *Computer Science Department, University of California*
Los Angeles CA 90024, U.S.A.
E-mail: zaniolo@cs.ucla.edu

The problem of finding effective logic-based formalizations for problems involving actions remains one of the main application challenges of non-monotonic knowledge representation. In this paper, we show that complex planning strategies find natural logic-based formulations and efficient implementations in the framework of deductive database languages. We begin by modeling classical STRIPS-like totally ordered plans by means of Datalog_{1S} programs, and show that these programs have a stable model semantics that is also amenable to efficient computation. We then show that the proposed approach is quite expressive and flexible, and can also model partially ordered plans, which are abstract plans whereby each plan stands for a whole class of totally ordered plans. This results in a reduction of the search space and a subsequent improvement in efficiency.

1 – Introduction

Planning has been among the first fields of computing to pursue the use of logic as a formal model of computation. Early attempts brought into focus two serious problems with the logic-based approach: One is the frame problem [23], and the other is the computational inefficiency of resolution-based theorem provers initially used to construct plans. Much progress has been made since then. In particular, the introduction of the concept of implicit negation and the supporting theory of non-monotonic reasoning allow the derivation of negative information by default, that is, without having to explicitly state negative facts. A survey of the large body of research on non-monotonic reasoning will take us beyond the scope of this paper. However, this paper benefits significantly from this body of work, particularly through the notion of stable models [16]. Indeed, this notion captures in a simple definition many of the key ideas underlying the different and more complex theories of non-monotonic reasoning [21].

Much progress in the logic-based approach has also been achieved on the computational front. In particular, the introduction of Prolog allowed searches for plans to be expressed as programs that execute efficiently on specialized SLD-resolution engines such as Prolog's WAM [32]. The usage of Prolog, however, is not completely satisfactory from a logical viewpoint, due to the many non-logical constructs used in the language and to the limited support for implicit negation provided by SLDNF [10].

A related area that has seen much recent progress is that of deductive databases. Deductive database languages are different from Prolog in their preference for a bottom-up (forward chaining) implementation approach, and their attempt to reconcile the declarative logic-based semantics with computational efficiency. Therefore, while deductive database systems do not support recursive rules with negation under the unrestricted stable model semantics (which is NP-hard to implement [17]), they support concepts such as stratified negation that characterize classes of programs for which stable models can be computed efficiently (i.e., in polynomial time). The *choice* construct [27] is critical in this context, since the combination of stratified negation and choice ensures P-time completeness (under genericity) of Datalog programs [1] (i.e., the expressibility of all functions that are computable in polynomial time on the size of the database). Choice is in fact only a short-hand for a restricted form of negation in recursive rules, where each stable model of a choice program can be computed in polynomial time [11, 27].

The key contribution of this paper is showing that complex planning strategies find natural logic-based formulations and efficient implementations in the framework of deductive database languages. Therefore, we begin by modeling classical STRIPS-like totally ordered plans using Datalog_{1S} rules (similar to those supported, e.g., in systems such as $\mathcal{LDL}$++ [36] or Aditi [31]). Then, we elaborate on the following two points:

(1) Because of their syntactic structure, the resulting planning rules always have a declarative semantics, based on the notion of stable model. Moreover, each stable model can be constructed in polynomial time.

(2) The proposed approach is quite expressive and flexible. We provide evidence of this by modeling partially ordered plans. Partially ordered plans are abstract plans whereby each plan stands for a whole class of equivalent STRIPS plans. As such, the search space can be restricted inasmuch as equivalent plans need not be tested (systematicity). We show that these and other improvements are easily incorporated into our Datalog_{1S} programs.

The organization of the paper is as follows. The basic notions of totally ordered (or linear) plan are introduced in the next section, whereas the logic-based model for totally ordered plans is introduced in Section 3. In Section 4 we show the existence of a stable model semantics for totally ordered plans. Furthermore, the special syntactic structure of the resulting programs (XY-stratification) allows each stable model to be efficiently constructed using a modified iterated fixpoint procedure [36]. In Section 5, we introduce the notion of partially ordered plan, where the notion of systematicity entails a compression of the search space, and we show that partially ordered plans can be easily included into our logic-based framework. A discussion of related work is reported in Section 6, while Section 7 contains some concluding remarks.

2 – Preliminaries

This section briefly introduces the definition of totally ordered plan, along with some terminology and notations which will be used in the rest of the paper.

A *planning problem* can be described by a triple $< S, G, \mathcal{A} >$, where S is a description of an initial state, G is the description of a goal, and $\mathcal{A}$ is a set of actions (also called "operators").

Following the `STRIPS` representation [15], a *state* is a finite set of (ground) atoms. Intuitively speaking, a state establishes which atoms are currently true according to the standard definition of satisfaction of first-order logic. If a ground atom A belongs to a state S ($A \in S$) then A is true in state S, while if $A \notin S$ then A is false in S (and $\neg A$ is true).

Each *action* α is a quadruple $< Name(\alpha), Pre(\alpha), Add(\alpha), Del(\alpha) >$ where:

- $Name(\alpha)$ is a syntactic expression of the form $\alpha(X_1, \ldots, X_n)$,
- $Pre(\alpha)$ is a finite set of literals, called *preconditions* of α, whose set of variables is $\{X_1, \ldots, X_n\}$, and
- $Add(\alpha)$ and $Del(\alpha)$ are finite sets of atoms, whose variables are all from the set $\{X_1, \ldots, X_n\}$. $Add(\alpha)$ is called the set of *additions* of α while $Del(\alpha)$ is called the set of *deletions* of α.

Notice that we allow negative literals of the form $\neg A$ to occur in action preconditions, as for instance in [12]. We will also denote by $Post(\alpha)$ the set of postconditions of α, that is the set $\{\neg x \mid x \in Del(\alpha)\} \cup \{x \mid x \in Add(\alpha)\}$.

A *goal* is a conjunction of atoms whose variables (if any) are existentially quantified (and we will often abuse notation and write goals as conjunctions of atoms without writing the quantifiers).

Let $\Pi =< S, G, \mathcal{A} >$ be a planning problem, let α be an action in $\mathcal{A}$ whose name is $\alpha(X_1, \ldots, X_n)$, and let ϑ be a substitution that assigns ground terms to each X_i, for $i \in \{1, \ldots, n\}$. We say that α is *ϑ-executable* in a state S, *resulting* in state S' if and only if:
(1) $S \models \{L\vartheta \mid L \in Pre(\alpha)\}$, and (2) $S' = (S \setminus (Del(\alpha)\vartheta)) \cup (Add(\alpha)\vartheta)$.
If α is *ϑ-executable* in a state S, *resulting* in state S', we also write $S \stackrel{\alpha,\vartheta}{\Longrightarrow} S'$.

A sequence of actions $< \alpha_1\vartheta_1; \ldots; \alpha_n\vartheta_n >$ will be called a *totally ordered plan* (of length n) for a planning problem $\Pi =< S, G, \mathcal{A} >$ if and only if: (1) $\{\alpha_1, \ldots, \alpha_n\} \subseteq \mathcal{A}$, and (2) there exists a sequence of states $S_1, \ldots, S_n$ such that: $S \stackrel{\alpha_1,\vartheta_1}{\Longrightarrow} S_1 \stackrel{\alpha_2,\vartheta_2}{\Longrightarrow} S_2 \ldots \stackrel{\alpha_n,\vartheta_n}{\Longrightarrow} S_n$.

A totally ordered plan $< \alpha_1\vartheta_1; \ldots; \alpha_n\vartheta_n >$ will be called a *successful plan* for $\Pi =< S, G, \mathcal{A} >$ (or simply a *solution* for Π) if there exists a ground instance of G which is true in S_n.

2.1 – Motivating Example

The blocks world is a well-known planning domain that was introduced with the `STRIPS` planner [15]. The standard blocks world domain contains the following predicates:

`hand_empty`:	The hand of the robot is empty
`holding(X)`:	The robot is holding block `X` in its hand
`on(X,Y)`:	Block `X` is on block `Y`
`ontable(X)`:	Block `X` is on the table
`clear(X)`:	There is nothing on top of block `X`

The following actions are used:

`pickup(X)`: Pick up block `X` from the table.
`Pre` = {`ontable(X), clear(X), hand_empty`}
`Post` = {¬`ontable(X)`, ¬`hand_empty, holding(X)`}

`putdown(X)`: Put block `X` on the table.
`Pre` = {`holding(X)`}
`Post` = {¬`holding(X), ontable(X), hand_empty` }

`stack(X,Y)`: Put block `X` on top of block `Y`.
`Pre` = {`clear(Y), holding(X)`}
`Post` = {¬`clear(Y)`, ¬`holding(X), on(X,Y), hand_empty`}

`unstack(X,Y)`: Remove block `X` from top of block `Y`.

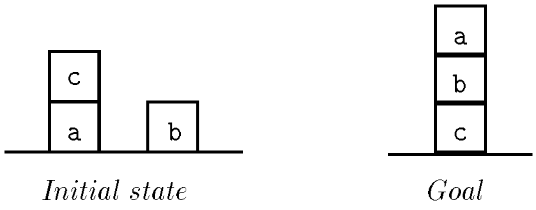

Figure 1: Blocks world example.

```
Pre  = {clear(X), on(X,Y), hand_empty}
Post = {¬on(X,Y), ¬hand_empty, clear(Y), holding(X)}
```

In the rest of the paper we will consider the following planning problem (Figure 1). Given the above actions, we consider an initial state in which there are three blocks `a`, `b`, and `c`. Blocks `a` and `b` are on the table, while block `c` is on top of `a`. The goal is to build a stack of blocks such that block `a` is on top of `b` which in turn is on top of block `c`.

For the sake of uniformity, according to [22, 33], the initial state and the goal of the planning problem are also represented by two special actions called `begin` and `end`, respectively. Action `begin` has a special precondition `start` and one postcondition for each literal that is true in the initial state. Action `end` has a special postcondition `done` and one precondition for each conjunct of the goal.

```
begin:
Pre  = {start}
Post = {¬start, hand_empty, clear(c), on(c,a), ontable(a),
        clear(b), ontable(b)}

end:
Pre  = {¬done, ontable(c), on(b,c), on(a,b)}
Post = {done}
```

3 – Modeling Totally Ordered Plans

We first show how a planning problem $\Pi = < S, G, \mathcal{A} >$ can be represented by means of a logic program $\mathcal{CH}(\Pi)$ with choice constructs [11, 27].

We will use Datalog$_{1S}$ to model state changes [8]. The merits of Datalog$_{1S}$ for modeling temporal and dynamic systems have been described for instance in [9, 34, 35]. In Datalog$_{1S}$ predicates may have an additional argument called the *stage argument*. Values in the stage argument are taken from the domain 0, 0+1, 0+1+1, ..., that is the domain of integers generated by using the postfix successor function +1. For instance, the integer 3 is represented as 0+1+1+1. Alternatively, using the standard functional notation, the successor of J can denoted by s(J), and this notation is at the root of the name Datalog$_{1S}$. A predicate p with stage argument J has therefore the form $\mathtt{p}(\mathtt{J}, \vec{\mathtt{t}})$. For notational convenience, we will write the stage argument as superscript, so that for instance $\mathtt{p}(\mathtt{J}, \vec{\mathtt{t}})$ will be written as $\mathtt{p}^{\mathtt{J}}(\vec{\mathtt{t}})$.

Action description

We first show how actions in a planning domain can be naturally translated into logic clauses. For each action $\alpha \in \mathcal{A}$ we define a clause of the form:

$$\mathtt{firable}^{\mathtt{J}}(\alpha) \quad \leftarrow \quad \mathtt{p}_1^{\mathtt{J}}(\vec{\mathtt{t}}_1), \ldots, \mathtt{p}_{\mathtt{m}}^{\mathtt{J}}(\vec{\mathtt{t}}_{\mathtt{m}}), \neg \mathtt{q}_1^{\mathtt{J}}(\vec{\mathtt{u}}_1), \ldots, \neg \mathtt{q}_{\mathtt{n}}^{\mathtt{J}}(\vec{\mathtt{u}}_{\mathtt{n}}). \tag{1}$$

where $Pre(\alpha) = \{\mathtt{p}_1(\vec{\mathtt{t}}_1), \ldots, \mathtt{p}_{\mathtt{m}}(\vec{\mathtt{t}}_{\mathtt{m}}), \neg \mathtt{q}_1(\vec{\mathtt{u}}_1), \ldots, \neg \mathtt{q}_{\mathtt{n}}(\vec{\mathtt{u}}_{\mathtt{n}})\}$. The meaning of this clause is that action α is firable (with a suitable instantiation of the variables in its name) at stage J if all its preconditions are true at stage J.

Observe that in the body of the rules we do not use function symbols, and that in the head of the rules we take advantage of the notational convenience of denoting an action by its name (such as pickup(X)). As we shall discuss in Section 4.2, these rules can be replaced by standard Datalog$_{1S}$ rules, where no function symbol is allowed besides that in the stage argument.

The initial state and the goal of the planning problem are also represented as actions, namely by the two special actions begin and end, as described in Section 2.1. The former is always the first action to be fired in any planning problem, and we therefore introduce the exit rule:

$$\mathtt{start}^{0}. \tag{2}$$

which states that the atom start is true at stage 0, and hence action begin will always be firable at the first stage.

Action selection

At each step of the computation one, and only one, action can be selected for execution among all firable actions. This behavior can be suitably expressed by means of the choice operator:

$$\begin{aligned} \mathtt{fired}^{\mathtt{J}}(\alpha) \quad \leftarrow \quad & \mathtt{firable}^{\mathtt{J}}(\alpha), \\ & \mathtt{choice}^{\mathtt{J}}(\alpha), \\ & \neg\mathtt{firable}^{\mathtt{J}}(\mathtt{end}). \end{aligned} \tag{3}$$

Clause (3) states that an instance of action α is fired at stage J if such an instance is firable at stage J and if the nondeterministic choice construct selects that instance among all the (instances of) actions firable at step J. The predicate choice is a special predicate that, for each instance of its first argument (J), nondeterministically picks up a unique instance of the second argument (α). The last condition in clause (3) ($\neg\mathtt{firable}^{\mathtt{J}}(\mathtt{end})$) enforces that an action can be fired at stage J only if the special action end is not firable at that stage. Finally:

$$\begin{array}{rcl} \mathtt{fired}^{\mathtt{J}}(\mathtt{end}) & \leftarrow & \mathtt{firable}^{\mathtt{J}}(\mathtt{end}), \\ & & \neg\mathtt{done}. \end{array} \tag{4}$$

Clause (4) states that the action end is fired as soon as it becomes firable. The second condition in the clause ($\neg$done) guarantees that end will be fired at most once.

Frame Axioms

We will represent action postconditions as a database relation of the form:

$$\mathtt{postcond}(\alpha, \gamma, \mathtt{Type}) \tag{5}$$

where α denotes an action, γ denotes a postcondition without the negation symbol, and Type is either pos, for positive postconditions, or neg for negative ones. Thus, for the example at hand we have:

($p1$) postcond(pickup(X), ontable(X), neg).
($p2$) postcond(pickup(X), hand_empty, neg).
($p3$) postcond(pickup(X), holding(X), pos).
($p4$) postcond(putdown(X), holding(X), neg).
($p5$) postcond(putdown(X), ontable(X), pos).
($p6$) postcond(putdown(X), hand_empty, pos).
($p7$) postcond(stack(X, Y), clear(Y), neg).
($p8$) postcond(stack(X, Y), holding(X), neg).
($p9$) postcond(stack(X, Y), on(X, Y), pos).
($p10$) postcond(stack(X, Y), hand_empty, pos).
($p11$) postcond(unstack(X, Y), on(X, Y), neg).
($p12$) postcond(unstack(X, Y), hand_empty, neg).
($p13$) postcond(unstack(X, Y), clear(Y), pos).
($p14$) postcond(unstack(X, Y), holding(X), pos).

The postcondition of begin and end are:

($p15$) postcond(begin, start, neg).
($p16$) postcond(begin, hand_empty, pos).
($p17$) postcond(begin, clear(c), pos).
($p18$) postcond(begin, on(c, a), pos).
($p19$) postcond(begin, ontable(a), pos).
($p20$) postcond(begin, clear(b), pos).
($p21$) postcond(begin, ontable(b), pos).
($p22$) postcond(end, done, pos).

State changes can be therefore modeled as follows:

$$\begin{array}{rcl} \mathtt{add}^{\mathtt{J}}(\mathtt{Cond}) & \leftarrow & \mathtt{fired}^{\mathtt{J}}(\alpha), \\ & & \mathtt{postcond}(\alpha, \mathtt{Cond}, \mathtt{pos}). \end{array} \tag{6}$$

$$\begin{array}{rcl} \mathtt{del}^{\mathtt{J}}(\mathtt{Cond}) & \leftarrow & \mathtt{fired}^{\mathtt{J}}(\alpha), \\ & & \mathtt{postcond}(\alpha, \mathtt{Cond}, \mathtt{neg}). \end{array} \tag{7}$$

$$\mathtt{p}^{\mathtt{J+1}}(\vec{\mathtt{t}}) \leftarrow \mathtt{add}^{\mathtt{J}}(\mathtt{p}(\vec{\mathtt{t}})). \tag{8}$$

$$\begin{array}{rcl} \mathtt{p}^{\mathtt{J+1}}(\vec{\mathtt{t}}) & \leftarrow & \mathtt{p}^{\mathtt{J}}(\vec{\mathtt{t}}), \\ & & \neg\mathtt{del}^{\mathtt{J}}(\mathtt{p}(\vec{\mathtt{t}})). \end{array} \tag{9}$$

The last two clauses play the role of "frame axioms" in this context. They state that if a new atom is added (resp., deleted) at stage J, then this atom becomes true (resp., false) at stage J+1.

Given a planning problem $\Pi =< S, \mathcal{A}, G >$, we can therefore translate Π into a choice logic program $\mathcal{CH}(\Pi)$ by:

(i) Associating a clause (1) with each action $\alpha \in \mathcal{A}$, and a clause (5) with each action postcondition,

(ii) Associating clauses (8) and (9) with each predicate p, and

(iii) Including verbatim the clauses (2),(3),(4), (6), and (7).

Example 1
We now show how the blocks world example discussed in Section 2.1 can be modeled by means of a choice logic program.

We first include the rules of type (1) to define the firable *predicate for each action:*

$$
\begin{array}{llcl}
(c1) & \texttt{firable}^{\texttt{J}}(\texttt{pickup(X)}) & \leftarrow & \texttt{ontable}^{\texttt{J}}(\texttt{X}), \texttt{clear}^{\texttt{J}}(\texttt{X}), \texttt{hand\_empty}^{\texttt{J}}. \\
(c2) & \texttt{firable}^{\texttt{J}}(\texttt{putdown(X)}) & \leftarrow & \texttt{holding}^{\texttt{J}}(\texttt{X}). \\
(c3) & \texttt{firable}^{\texttt{J}}(\texttt{stack(X,Y)}) & \leftarrow & \texttt{clear}^{\texttt{J}}(\texttt{Y}), \texttt{holding}^{\texttt{J}}(\texttt{X}). \\
(c4) & \texttt{firable}^{\texttt{J}}(\texttt{unstack(X,Y)}) & \leftarrow & \texttt{clear}^{\texttt{J}}(\texttt{X}), \texttt{on}^{\texttt{J}}(\texttt{X},\texttt{Y}), \texttt{hand\_empty}^{\texttt{J}}. \\
(c5) & \texttt{firable}^{\texttt{J}}(\texttt{begin}) & \leftarrow & \texttt{start}^{\texttt{J}}. \\
(c6) & \texttt{firable}^{\texttt{J}}(\texttt{end}) & \leftarrow & \neg\texttt{done}^{\texttt{J}}, \texttt{ontable}^{\texttt{J}}(\texttt{c}), \texttt{on}^{\texttt{J}}(\texttt{a},\texttt{b}), \texttt{on}^{\texttt{J}}(\texttt{b},\texttt{c}).
\end{array}
$$

The frame axioms lead to the following clauses:

$$
\begin{array}{llcl}
(c7) & \texttt{hand\_empty}^{\texttt{J+1}} & \leftarrow & \texttt{add}^{\texttt{J}}(\texttt{hand\_empty}). \\
(c8) & \texttt{hand\_empty}^{\texttt{J+1}} & \leftarrow & \texttt{hand\_empty}^{\texttt{J}}, \neg\texttt{del}^{\texttt{J}}(\texttt{hand\_empty}). \\
(c9) & \texttt{holding}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{add}^{\texttt{J}}(\texttt{holding(X)}). \\
(c10) & \texttt{holding}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{holding}^{\texttt{J}}(\texttt{X}), \neg\texttt{del}^{\texttt{J}}(\texttt{holding(X)}). \\
(c11) & \texttt{on}^{\texttt{J+1}}(\texttt{X},\texttt{Y}) & \leftarrow & \texttt{add}^{\texttt{J}}(\texttt{on(X,Y)}). \\
(c12) & \texttt{on}^{\texttt{J+1}}(\texttt{X},\texttt{Y}) & \leftarrow & \texttt{on}^{\texttt{J}}(\texttt{X},\texttt{Y}), \neg\texttt{del}^{\texttt{J}}(\texttt{on(X,Y)}). \\
(c13) & \texttt{ontable}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{add}^{\texttt{J}}(\texttt{ontable(X)}). \\
(c14) & \texttt{ontable}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{ontable}^{\texttt{J}}(\texttt{X}), \neg\texttt{del}^{\texttt{J}}(\texttt{ontable(X)}). \\
(c15) & \texttt{clear}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{add}^{\texttt{J}}(\texttt{clear(X)}). \\
(c16) & \texttt{clear}^{\texttt{J+1}}(\texttt{X}) & \leftarrow & \texttt{clear}^{\texttt{J}}(\texttt{X}), \neg\texttt{del}^{\texttt{J}}(\texttt{clear(X)}).
\end{array}
$$

The program $\mathcal{CH}(\Pi)$ therefore consists of clauses (c1)—(c16), together with clauses (2),(3),(4), (6), (7), and of the set of facts $(p1)-(p22)$ of type (5) previously introduced.

4 – Formal Semantics

Using the techniques described in [27], any choice logic program P can be converted into a logic program with negation denoted as $foe(P)$. In particular, in a program $\mathcal{CH}(\Pi)$ representing a planning problem Π, clause (3) can be converted into the following set of clauses:

$$
\begin{array}{rcl}
\texttt{fired}^{\texttt{J}}(\alpha) & \leftarrow & \texttt{firable}^{\texttt{J}}(\alpha), \\
& & \texttt{chosen}^{\texttt{J}}(\alpha), \\
& & \neg\texttt{firable}^{\texttt{J}}(\texttt{end}).
\end{array} \tag{10}
$$

$$
\begin{array}{rcl}
\texttt{chosen}^{\texttt{J}}(\alpha) & \leftarrow & \texttt{firable}^{\texttt{J}}(\alpha), \\
& & \neg\texttt{diffchoice}^{\texttt{J}}(\alpha).
\end{array} \tag{11}
$$

$$
\begin{array}{rcl}
\texttt{diffchoice}^{\texttt{J}}(\alpha) & \leftarrow & \texttt{chosen}^{\texttt{J}}(\beta), \\
& & \alpha \neq \beta.
\end{array} \tag{12}
$$

Given a planning problem Π, we therefore denote by $foe(\mathcal{CH}(\Pi))$ the choice-free logic program obtained by:

(i) Associating a clause (1) with each action $\alpha \in \mathcal{A}$, and a clause (5) with each action postcondition,

(ii) Associating clauses (8) and (9) with each predicate `p`, and

(iii) Including verbatim clauses (2), (4), (6), (7), (10), (11), and (12).

Example 2
Let Π be the blocks world planning problem described in Section 2.1.
Then $foe(\mathcal{CH}(\Pi))$ contains all the clauses in $\mathcal{CH}(\Pi)$ except for clause (3) which is replaced by clauses (10), (11), and (12).

Let P be a choice logic program, and let $foe(P)$ be the corresponding choice-free logic program. We call $det(P)$ the program obtained from $foe(P)$ by:

(1) Removing all the clauses defining `diffchoice`, that is all the clauses having `diffchoice` as the predicate symbol in the head of the clause, and

(2) Removing all (positive and negative) literals of the form $\mathtt{diffchoice}^{\mathtt{J}}(\_)$ from the body of the remaining clauses.

We will call $det(P)$ the *deterministic version* of P.

Example 3
Let us consider again the blocks world example. In this case, $det(\mathcal{CH}(\Pi))$ consists of all the clauses in $foe(\mathcal{CH}(\Pi))$, except for clause (12) which is deleted, and for clause (11) that is replaced by the clause:

$$\mathtt{chosen}^{\mathtt{J}}(\alpha) \quad \leftarrow \quad \mathtt{firable}^{\mathtt{J}}(\alpha). \tag{13}$$

which is obtained from clause (11) by deleting the negated `diffchoice` *atom in its body.*

Definition 1
Let P be a logic program with negated goals and choice goals. We say that P is locally stratified modulo choice *if and only if the deterministic version of P, $det(P)$ is locally stratified.*

Proposition 2
For any planning domain Π, $det(\mathcal{CH}(\Pi))$ is locally stratified, and hence $\mathcal{CH}(\Pi)$ is locally stratified modulo choice.

Proof
Consider the dependency graph for the program $foe(\mathcal{CH}(\Pi))$ corresponding to a planning problem Π. It is easy to observe that $foe(\mathcal{CH}(\Pi))$ is not locally stratified because of clauses (11) and (12) which introduce a cycle in the dependency graph. The transformation from $foe(\mathcal{CH}(\Pi))$ into $det(\mathcal{CH}(\Pi))$ actually eliminates the cycle by deleting the two edges connecting `chosen` and `diffchoice`. The program $det(\mathcal{CH}(\Pi))$ is therefore locally stratified. □

Given a planning problem Π, we can convert Π into a logic program $foe(\mathcal{CH}(\Pi))$ by first converting Π into a choice logic program $\mathcal{CH}(\Pi)$, and then transforming the latter into a choice-free logic program $foe(\mathcal{CH}(\Pi))$ that may contain negation in clause bodies.

It is well-known that a locally stratified logic program Q has a unique stable model, which can be computed as follows.

Iterated Fixpoint
Base
$M^0 = \emptyset$
Induction
For $j > 0$, M^j = the unique perfect model of $(M^{j-1} \cup ground^j(Q))$
where $ground^j(Q)$ denotes the clauses of $ground(Q)$ with head in the j-th stratum.

The result of the iterated fixpoint procedure:

$$M^\omega = \bigcup_{0 \leq j < \omega} M^j$$

coincides with the perfect model of Q, which is also the unique stable model of Q [26].

Consider now a locally stratified modulo choice program P. The program $foe(P)$ can be partitioned into two sets of clauses: A set of clauses $dfc(foe(P))$ defining the `diffchoice` predicate, and the remaining clauses $foe'(P) = foe(P) - dfc(foe(P)$. Then the rules in $ground(foe'(P))$ can be partitioned according to the strata of $det(P)$ to which their head belongs. Let $ground^j(foe'(P))$ denote the set of rules in $ground(foe'(P))$ whose head belong to the j-th stratum of $det(P)$. In the presence of choice, the iterated fixpoint computation can be extended as follows:

Iterated Choice Fixpoint
Base
$M^0 = \emptyset$
Induction
For $j > 0$, M^j = a stable model of $(M^{j-1} \cup ground^j(foe'(P)) \cup dfc(foe(P))$.

Since at each step there may be several stable models (each corresponding to a choice model for that stratum), we have a non-deterministic computation. Each result of such computation:

$$M^\omega = \bigcup_{0 \leq j < \omega} M^j$$

will be called *a result of the Iterated Choice Fixpoint on P.*

Definition 3
Let P be a choice logic program. An interpretation I of $foe(P)$ will be said to violate the constraint $\texttt{choice}^{\texttt{X}}\texttt{(Y)}$ *when I contains two* $\texttt{chosen}^{\texttt{X}}\texttt{(Y)}$ *atoms that are identical in their X-arguments but differ in their Y-arguments.*

When P is a positive program, then each stable model for $foe(P)$ violates none of the choice constraints of P. Now:

Lemma 4
Let P be a program that is locally stratified modulo choice and M be a result of the Iterated Choice Fixpoint on P. Then, M violates none of the choice constraints of P.

Proof
By contradiction, let c_1 and c_2 denote two *chosen* atoms in M that violate a choice constraint. If c_1 and c_2 have been produced at the same step, j, then M^j cannot be a stable model. Then, say that c_1 and c_2 have been produced, respectively, at steps j_1 and j_2, with $j_1 < j_2$. In this case, for each rule with head c_2, the negated $diffchoice$ goal in this rule is false. Thus M^{j_2} is not a minimal model, and neither can it be a stable one. □

We can now prove the soundness and non-deterministic completeness of the iterated choice fixpoint. (Let $ST(P, N)$ denote the program $ground_N(P)$, i.e., the positive ground program produced from $ground(P)$ by model N, according to the stability transformation described in [16].)

Theorem 5 (Soundness)
Let P be a program that is locally stratified modulo choice. Then, each result of the iterated choice fixpoint on P is a stable model for $foe(P)$.

Proof

Let N be a result of the choice fixpoint computation. Since it satisfies every rule in $ground(foe(P))$, N is a model of $foe(P)$. Let M be the minimal model of $ground_N(foe(P)) = ST(foe(P), N)$. To prove stability, we need to show that $N \subseteq M$. To this end, we use the property that the rules in a positive program can be fired in any order. Thus we can proceed by firing at each stratum j all the firable rules, in $ground^j_N(foe'(P))$ along with those in $ground(dfc(foe(P)))$. Thus, the minimal model of $ground_N(foe(P))$ can be computed by first computing $M^0 = \emptyset$ and then computing M^j as the minimal model of $M^{j-1} \cup ground^j_N(foe'(P)) \cup dfc(foe(P))$, for $j > 0$. Let N^j denote the atoms in N belonging either to the j-th stratum of $det(P)$ or the lower ones. We need to show that $N^j \subseteq M^j$, for $0 \leq j < \omega$. The property holds for $j = 0$. By contradiction, say that there exists a positive integer k, such that the property holds for every $j < k$ but not for k. If so, then there must exist a rule $r \in ground^k(foe'(P)) - ground^k_N(foe'(P))$. Thus, r was eliminated by the stability transformation, due to the fact that N contains (i) a negated diffchoice goal of r or (ii) another negated goal of r. Now, if (i) holds then N violates the choice constraints, a contradiction according to the last lemma. Neither can (ii) hold, since this would contradict the assumption that $det(P)$ is locally stratified. This completes our proof. □

Theorem 6 (Completeness)
Let P be a program that is locally stratified modulo choice, and let N be stable model for $foe(P)$. Then N is also a result of the iterated choice fixpoint on P.

Proof

For N and P defined as in the theorem, let N^j denote the set of atoms in N belonging to the j-th stratum of $det(P)$ or the previous ones. To complete our proof we only need to show that, for $0 \leq j < \omega$, N^j is a stable model for $P^j = N^{j-1} \cup ground^j(foe'(P)) \cup dfc(foe(P))$. The property holds for $j = 0$. Now say that the property holds for $0 \leq j < k$, but not for k. Observe that if N^k is not a model for P^k, then every rule with head in $ground^k(foe'(P))$ that is not satisfied in N^k cannot be satisfied in N either, since their negated goals are from lower strata (and a similar conclusion holds for the rules in $dfc(foe(P))$, since these are positive). Thus, N^k must be a model for P^k. Now, by contradiction, say that some atom $a \in N^k$ is not in the minimal model of the program $ST(P^j, N^j)$. Then, neither can this atom a be in the minimal model of $ST(P^k, N)$, since $N \supseteq N^k$. Finally, since all the rules with head in a are in $ground^k(foe'(P))$, a cannot be in the program $ST(foe(P), N)$. This contradicts the assumption that N is a stable model and concludes our proof. □

4.1 – Properties of Stable Models

In this section we formalize the intuitive relationships that exist between stable models of the logic program $foe(\mathcal{CH}(\Pi))$ and the original planning problem Π.

A first simple property of the stable models of $\mathcal{CH}(\Pi)$ shows that every stable model of $foe(\mathcal{CH}(\Pi))$ causes at most one action to be executed at any given point in time.

Proposition 7
Let Π be a planning problem, and let M be a stable model of $foe(\mathcal{CH}(\Pi))$. Then for any integer $\mathtt{J} \geq 0$, there is at most one atom in M of the form $\mathtt{fired}^{\mathtt{J}}(\_)$.

Proof
Suppose that for some $\mathtt{J} \geq 0$ there are two atoms $\mathtt{fired}^{\mathtt{J}}(\beta_1)$, $\mathtt{fired}^{\mathtt{J}}(\beta_2)$ in M, with $\beta_1 \neq \beta_2$. Since all stable models are supported models [20] then, by clause (10), also $\mathtt{chosen}^{\mathtt{J}}(\beta_1)$ and $\mathtt{chosen}^{\mathtt{J}}(\beta_2)$ are in M. Hence, by clause (11) and since M is supported, we have that $\mathtt{diffchoice}^{\mathtt{J}}(\beta_1), \mathtt{diffchoice}^{\mathtt{J}}(\beta_2)$ are not in M. Since, $\beta_1 \neq \beta_2$, it follows from clause (12) that either $\mathtt{chosen}^{\mathtt{J}}(\beta_1)$ or $\mathtt{chosen}^{\mathtt{J}}(\beta_2)$ is not in M. Contradiction. □

The following proposition states that if M is a stable model, and α is *any* action firable at stage $\mathtt{J}$, then there is a stable model M' that is exactly like M up to stage $\mathtt{J} - \mathtt{1}$ included, but which causes action α to be fired at stage $\mathtt{J}$.

Proposition 8
Let Π be a planning problem, and let M be a stable model of $foe(\mathcal{CH}(\Pi))$. Let $[M]_{\mathtt{J}} = \{\mathtt{q}^{\mathtt{I}}(\vec{\mathtt{t}}) \mid \mathtt{q}^{\mathtt{I}}(\vec{\mathtt{t}}) \in M \text{ and } \mathtt{I} < \mathtt{J}\}$. Let α be any action such that $\exists\vartheta$ $\mathtt{firable}^{\mathtt{J}}((\alpha)\vartheta) \in M$, but $\mathtt{fired}^{\mathtt{J}}((\alpha)\vartheta) \notin M$. Then there is a stable model M' of $foe(\mathcal{CH}(\Pi))$ such that $[M]_{\mathtt{J}} = [M']_{\mathtt{J}}$ and such that $\mathtt{fired}^{\mathtt{J}}((\alpha)\vartheta) \in M'$.

Proof
Immediate consequence of the construction of $foe(\mathcal{CH}(\Pi))$, since for each firable action at stage $\mathtt{J}$ there is a stable model of $foe(\mathcal{CH}(\Pi))$ in which such an action is the $\mathtt{chosen}$ one. □

It turns out that there is a close correspondence between the existence of a solution for a planning problem and the existence of a stable model for the corresponding logic program. The following result explicitly states such a connection.

Proposition 9
Let Π *be a planning problem.*

$$\pi =< (\alpha_1)\vartheta_1; \ldots; (\alpha_n)\vartheta_n > \text{ is a solution for } \Pi$$
$$\Longleftrightarrow$$
$$\exists \text{ a stable model } M \text{ of } foe(\mathcal{CH}(\Pi)) \text{ s.t.}$$
$$\{\mathtt{fired^0(begin)}, \mathtt{fired^1}((\alpha_1)\vartheta_1), \ldots, \mathtt{fired^n}((\alpha_n)\vartheta_n), \mathtt{fired^{n+1}(end)}\vartheta_{n+1}\} \subseteq M.$$

The above proposition shows that any planning problem Π can be converted into a logic program with non-monotonic negation such that a solution for Π exists if and only if a corresponding set of atoms is true in some stable model of the corresponding logic program. Furthermore, the conversion of the planning domain into such a logic program can be performed in linear-time.

One of the interesting questions in planning is how to determine whether a given sequence of actions is a solution for a planning problem Π. Simply stated, the validity of a sequence of actions for a planning problem Π corresponds to the existence of a stable model for the program $foe(\mathcal{CH}(\Pi))$ suitably extended so as to represent that the actions in the sequence were fired.

Corollary 10
Let Π *be a planning problem, let* $\pi =< (\alpha_1)\vartheta_1; \ldots; (\alpha_n)\vartheta_n >$ *be a sequence of actions, and let*

$$\Delta = \{\mathtt{chosen^0(begin)}, \mathtt{chosen^1}((\alpha_1)\vartheta_1), \ldots, \mathtt{chosen^n}((\alpha_n)\vartheta_n), \mathtt{chosen^{n+1}}((\mathtt{end})\vartheta_{n+1})\}.$$

Then:

$$\pi \text{ is a solution for } \Pi \iff foe(\mathcal{CH}(\Pi)) \cup \Delta \text{ has a stable model.}$$

4.2 – Datalog$_{1S}$ *and* XY*-Stratification*

In the previous sections we have ensured the existence of a declarative semantics based on stable models for our planning problem. However, the existence of such declarative semantics hardly guarantees any efficient implementation. In fact, we know that the iterated choice fixpoint procedure can only be used after local stratification is known — an undecidable problem for general *Datalog* with function symbols [1]. The formalization proposed in this paper, however, is amenable to efficient implementation, as demonstrated by the following analysis based on the two important notions of Datalog$_{1S}$ and XY-Stratification [36].

In Section 2, the name of an action is defined as a syntactic expression of the form *name(list-of-parameters)*, such as `stack(X,Y)`. Since strict Datalog$_{1S}$ does not allow the use of function symbols (other than the successor function `s`), a formula of the form $\texttt{firable}^{\texttt{J}}(\texttt{stack}(\texttt{X},\texttt{Y}))$ should be represented as $\texttt{firable}^{\texttt{J}}(\texttt{stack},\texttt{X},\texttt{Y})$. In order to represent all actions with the same number n of terms, $n-1$ can be taken as the largest number of parameters of the actions in the domain. In the representation of names of actions with m parameters, where $m < n$, only the first $m+1$ arguments will be therefore significant. For instance, in the blocks world example of Section 2.1, the largest number of parameters of an action is 2, as in `stack(X,Y)`. The representation of actions with less than 2 parameters, such as `pickup(X)`, has therefore the form `(pickup, X, nil)`, where `nil` is an arbitrary distinguished constant. In the rest of the paper (especially in the examples), for the sake of readability, we will however abuse notation and write action names in the form `stack(X,Y)`.

In summary, a planning problem Π can be modeled by a Datalog$_{1S}$ program $\mathcal{CH}(\Pi)$. Consider now the program P' obtained by removing the $1S$ argument from our program. P' is a Datalog program without any function symbols, thus it has a finite Herbrand Base which we will denote by $B'_{\mathcal{CH}(\Pi)}$, and call it the *reduced Herbrand base* for our program.

An important property of our planning programs is that they have a particular syntactic structure whereby the local strata can be determined at compile time [36]. The structure follows from the interpretation of the heads of recursive Datalog$_{1S}$ rules as defining new-values for the predicates. Goals of the recursive rules that have the same stage values as those in the head thus refer to the new values of the predicates, while those goals with a stage argument one less than that of the head refer to the old values of the predicates. For a set of recursive rules to be XY-stratified [36]: (i) no recursive goal, other than the 'old' and 'new' ones just described are allowed in the rules, and (ii) the *primed predicate dependency graph* must be acyclic [36]. The primed dependency graph shows the dependencies between the 'old' and 'new' instances of each predicate name, where the 'new' names have been primed to distinguish them from the 'old' ones. For the example at hand, for instance, the primed dependency graph is shown in Figure 2.

XY-stratified programs have a natural local stratification, resulting from the partition of the atoms in the Herbrand universe into:
(i) major strata, according to their stage arguments; furthermore, each major stratum is refined into
(ii) minor strata as per the stratification of the primed dependency graph.

Thus, for programs that are XY-stratified, the iterated fixpoint is no longer transfinite but it proceeds in successive steps, according to the stage values and to the order induced by the primed dependency graph. In other words, having completed the computation of all the atoms with stage value J, these are now regarded as old values. We can proceed with the computation of atoms with stage value $J+1$

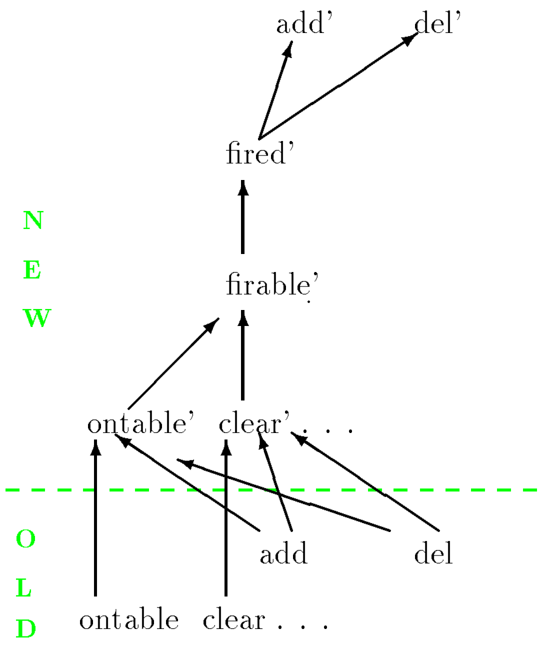

Figure 2: Primed dependency graph for the blocks world example.

in the bottom-up order defined by the Figure 2. Thus, the frame axiom rules are fired first, yielding new predicate values. Then the `firable` predicates are computed next, and the actual `fired` are computed from these. The new `del` and `add` predicates are computed last (for the stage value $J+1$). Then the computation of stage value $J+2$ will start in a similar fashion. Thus the "computation of stable models" in the Iterated Choice Fixpoint simplifies dramatically, for programs that are XY-stratified modulo choice. For rules (1), (2), (5), (6), (7), and (8) there is a single computation step of T_P. The computation of the choice rule (3) for each stratum is also very simple. This rule is now non-recursive, and J is constant within this stratum. Thus the stable model for this stratum is found once the first successful instance of this rule is computed.

Now observe that the XY-stratification holds both for $\mathcal{CH}(\Pi)$ and its Datalog$_{1S}$ counterpart. Then the following result follows immediately from the definitions.

Lemma 11
Let $\mathcal{CH}(\Pi)$ be the program associated with a planning problem Π and let $B'_{\mathcal{CH}(\Pi)}$ be the reduced Herbrand base of its Datalog$_{1S}$ counterpart. Then the following properties hold:
(i) $\mathcal{CH}(\Pi)$ is XY-stratified modulo choice,
(ii) $\mathcal{CH}(\Pi)$ has one or more stable models,
(iii) Each such stable model is computable in time which is linear in the length of the plan represented by the stable model, and is polynomial in $N = |B'_{\mathcal{CH}(\Pi)}|$, i.e., the size of the the reduced Herbrand Base of $\mathcal{CH}(\Pi)$.

Thus the construction of each stable model representing a plan can be performed in time which is linear in the length of the plan and polynomial with respect to the size of the planning problem $N = |B'_{\mathcal{CH}(\Pi)}|$. Due to the many efficiency improvements connected with XY-stratification, the coefficient and degree of the polynomial can in fact be kept very small, and the construction of single plans is done quite efficiently by systems such as $\mathcal{LDL}$++ and Aditi. Nevertheless, the number of alternative plans (i.e., choice models) will normally grow exponentially with the length of the plans. The notions of partially ordered plan and systematicity discussed in the next section can be used to reduce such a growth.

5 – Partially Ordered Plans

In the previous sections we have described a minimalist approach to modeling planning problems that favorably compares with previous approaches from the viewpoints of logical simplicity and amenability to efficient implementation. A natural question (which will be further discussed in Sections 6 and 7) concerns the extensibility and robustness of the proposed approach. Concretely, this question can only be answered by testing the approach on planning frameworks that are more advanced and complex than STRIPS. In this section, we show how the approach presented in the previous sections can be extended to model partially ordered plans. Its application to parallel plans is discussed in [5].

We first introduce briefly the definition of partially ordered plan. As discussed in Section 2, a totally ordered plan is a sequence of actions $<\alpha_1;\ldots;\alpha_n>$ which are totally ordered. Namely, if $i<j$ then α_i is executed before α_j in the plan. Therefore, totally ordered plans are generally over-constrained in the sense that they always define a total order, whereas only a partial order is needed for semantic reasons.

NOAH [28] and NONLIN [30] were the first planners based on the idea that, during the search for a plan, ordering constraints should be imposed between actions only if these are necessary for the achievement of the overall plan. The result of the planning process is therefore a *partially ordered* plan, that is a plan in which actions are partially rather than totally ordered. A partially ordered plan therefore represents a set of totally ordered plans.

5.1 – Causal Links

Partially ordered plans can be viewed as an abstraction of totally ordered plans, where only the essential ordering constraints due to causal links are represented. The notion of causal link was introduced by Tate [30] in the NONLIN planner as a means for representing the causal dependencies between actions in a plan. A causal link can be represented by a triple $<\alpha, x, \beta>$ where α and β are actions and x is a literal which is both in the postconditions of α (the producer of x) and in the preconditions of β (the consumer of x). Causal links are also written as $\alpha \xrightarrow{x} \beta$.

Definition 12
Let $\pi = <\alpha_1;\ldots;\alpha_n>$ be a totally ordered plan. The set $\mathcal{CL}(\pi)$ of causal links of π is defined as follows:

$$\begin{aligned}\mathcal{CL}(\pi) = \{\alpha_i \xrightarrow{x} \alpha_{i+j} \ \mid\ & x \in Post(\alpha_i) \ \wedge\ x \in Pre(\alpha_{i+j}) \ \wedge \\ & \not\exists \alpha_h :\ (x \in Post(\alpha_h) \ \wedge\ i < h < i+j)\}.\end{aligned}$$

If $\mathcal{CL}(\pi)$ contains a causal link $\alpha \xrightarrow{x} \beta$, we also say that α *causes* β in π (by means of x). Observe that if $\alpha \xrightarrow{x} \beta$ is a causal link in $\mathcal{CL}(\pi)$ then α is the *last* producer of x for β in π.

On the other hand, actions can also *inhibit* each other. We say that an action *inhibits* another action if the former makes false some of the preconditions of the latter, as formalized by the following definition.

Definition 13
Let α and β be two actions. We say that α inhibits *β if and only if*

$$\exists x : \ x \in Pre(\beta) \ \wedge \ \neg x \in Post(\alpha).$$

Namely α inhibits β if either α deletes one of the positive preconditions of β ($x \in Del(\alpha) \wedge x \in Pre(\beta)$), or if α adds one of the negative preconditions of β ($x \in Add(\alpha) \wedge \neg x \in Pre(\beta)$). (Notice that in Definition 13 and in the sequel we use the notational convention that $\neg(\neg x) = x$.)

Following [7], we now introduce a relation $\prec$ that defines a partial order on actions modeling the partially ordered abstraction of a plan. Intuitively speaking, $\alpha \prec \beta$ holds if the action α must precede β in the plan.

Definition 14
Let $\pi =< \alpha_1; \ldots; \alpha_n >$ be a totally ordered plan, and let α_i, α_{i+h} be two actions in π. The set of constraints $Ab(\pi)$ is defined as follows:

$$\alpha_i \prec \alpha_{i+h} \in Ab(\pi) \quad \textit{iff} \quad (1)\ \alpha_i \textit{ causes } \alpha_{i+h}, \textit{ or } (2)\ \alpha_{i+h} \textit{ inhibits } \alpha_i.$$

The partially ordered abstraction $Ab(\pi)$ of a plan π represents a set of totally ordered plans, namely the linearizations of $Ab(\pi)$.

Definition 15
Let $\pi =< \alpha_1; \ldots; \alpha_n >$ be a totally ordered plan.
A linearization *of $Ab(\pi)$ is a permutation $< \beta_1; \ldots; \beta_n >$ of $< \alpha_1; \ldots; \alpha_n >$ that satisfies $Ab(\pi)$ — i.e., such that $\not\exists i, h : \ \beta_{i+h} \prec \beta_i \in Ab(\pi)$.*

It is worth observing that the linearizations of a set of constraints $Ab(\pi)$ can be determined by applying standard algorithms on graphs. Indeed a set of constraints $Ab(\pi)$ can be represented by an acyclic graph where the nodes are the actions in π, and where there is a directed arc from node α to node β if and only if $\alpha \prec \beta \in Ab(\pi)$. It is easy to see that the linearizations of $Ab(\pi)$ simply correspond to topological sorts for the corresponding graph.

We will next introduce the notion of *strict actions* which simplifies the treatment of partially ordered plans without losing generality. As pointed out in [7], the *non-strict* interpretation of the semantics of deletions and additions as set-theoretic difference and union may generate *null* actions, that is actions whose execution does not modify the state. Under a strict definition of actions instead, each atom in the delete set of an action must also belong to its positive precondition set, so as to ensure that no deletion will be interpreted as a null action. The symmetric observation obviously holds for additions. Roughly speaking, an action that deletes x may be executed in a state S only if x belongs to S. Analogously an action that adds x may be executed in a state S only if x does not belong to S.

Definition 16
An action α is strict *if and only if* $\forall x : \ (x \in Post(\alpha) \Longrightarrow \neg x \in Pre(\alpha))$.

Namely an action α is strict if $\forall x : \ ((x \in Del(\alpha) \Longrightarrow x \in Pre(\alpha)) \wedge (x \in Add(\alpha) \Longrightarrow \neg x \in Pre(\alpha)))$. Note that the syntactic condition of strictness does not cause a loss of generality, since the non-strict interpretation of actions can be expressed by means of strict actions. For instance, suppose that α is not strict since there is a literal x such that $x \in Post(\alpha)$ and $\neg x \notin Pre(\alpha)$. Then α can be transformed into two strict actions by adding $\neg x$ to $Pre(\alpha)$, and by moving x from $Post(\alpha)$ into $Pre(\alpha)$, respectively. Therefore, we will employ a strict interpretation of actions and refer to strict plans and planning problems in the obvious way. Then we have the following important results.

Proposition 17
Let π be a strict solution for $\Pi =< S, \mathcal{A}, G >$. Then, each linearization of $Ab(\pi)$ is a solution for $\Pi =< S, \mathcal{A}, G >$.

Proof
We first show that each linearization π_1 of $Ab(\pi)$ is a totally ordered plan for Π. Let $\pi =< \alpha_1; \ldots; \alpha_n >$ and suppose that there is a linearization $\pi_1 =< \beta_1; \ldots; \beta_n >$ of $Ab(\pi)$ which is not a totally ordered plan for Π. Then there exists S_{i-1} such that: $S_0 \stackrel{\beta_1}{\Longrightarrow} S_1 \stackrel{\beta_2}{\Longrightarrow} S_2 \ldots \stackrel{\beta_{i-1}}{\Longrightarrow} S_{i-1}$, and $S_{i-1} \not\models Pre(\beta_i)$, i.e., $\exists L : \ L \in Pre(\beta_i) \wedge S_{i-1} \not\models L$. Since π_1 is a linearization of $Ab(\pi)$ there exists h: $\alpha_h = \beta_i$, and by strictness, there exists k: $(\alpha_{h-k} \stackrel{L}{\rightarrow} \alpha_h) \in \mathcal{CL}(\pi)$. By definition of $Ab(\pi)$, there exists j: $\alpha_{h-k} = \beta_{i-j}$. Then, since $S_{i-1} \not\models L$, there exists z such that $z \in (i-j, i)$ and $\neg L \in Post(\beta_z)$. Now, there exists $m \in [1, n]$: $\beta_z = \alpha_m$. We observe that $m \notin [h-k, h]$ since $z \in (i-j, i)$ and by strictness since $\alpha_{h-k} \stackrel{L}{\rightarrow} \alpha_h \in \mathcal{CL}(\pi)$. Moreover $m \not< h-k$ (otherwise $\beta_z \prec \beta_{i-j} \in Ab(\pi)$), and $m \not> h$ (otherwise $\beta_i \prec \beta_z \in Ab(\pi)$). Hence there is no $m \in [1, n]$: $\beta_z = \alpha_m$. Contradiction.
Finally if $\pi_1 =< \beta_1; \ldots; \beta_n >$ is a permutation of $\pi =< \alpha_1; \ldots; \alpha_n >$ and both π and π_1 are totally ordered plans for Π, then — by strictness — they produce the

same final state. Hence each linearization of $Ab(\pi)$ is also a solution for Π. $\square$

Proposition 18
Let π_1 be a strict totally ordered plan and let π_2 be a linearization of $Ab(\pi_1)$. Then:

$$Ab(\pi_1) = Ab(\pi_2).$$

Proof
We first show that $Ab(\pi_1) \subseteq Ab(\pi_2)$. Let $\alpha \prec \beta \in Ab(\pi_1)$. If β inhibits α then $\alpha \prec \beta \in Ab(\pi_2)$. If α causes β by means of some literal L then suppose that $\alpha \prec \beta \notin Ab(\pi_2)$. Then, by strictness, there exist γ and δ such that: $\pi_2 =< \ldots;\alpha;\ldots;\delta;\ldots;\gamma;\ldots;\beta;\ldots >$ where $L \in Post(\gamma)$ and $\neg L \in Post(\delta)$. Suppose now that δ precedes α in π_1. Then δ causes α in π_1, $\alpha \prec \beta \in Ab(\pi_1)$, and hence π_2 is not a linearization of $Ab(\pi_1)$. Contradiction. Suppose then that β precedes δ in π_1. Then, since δ inhibits β, $\beta \prec \delta \in Ab(\pi_1)$, and hence π_2 is not a linearization of $Ab(\pi_1)$. Contradiction.
We now show that $Ab(\pi_2) \subseteq Ab(\pi_1)$. Suppose that $\exists \alpha \prec \beta : \ \alpha \prec \beta \in Ab(\pi_2) \ \wedge \ \alpha \prec \beta \notin Ab(\pi_1)$. By definition of $Ab(\pi_2)$, $\alpha \prec \beta \in Ab(\pi_2)$ iff either α causes β in π_2 or β inhibits α. Since π_2 is a linearization of $Ab(\pi_1)$ then, by definition of causal link, $\mathcal{CL}(\pi_1) = \mathcal{CL}(\pi_2)$. Therefore if α causes β in π_2 then $\alpha \prec \beta \in Ab(\pi_1)$. Suppose then that β inhibits α on some literal L. If α precedes β also in π_1 then $\alpha \prec \beta \in Ab(\pi_1)$ and we have a contradiction. Suppose then that β precedes α in π_1. Then, by strictness, there exist $\delta_1, \ldots, \delta_{h+1}, \gamma_1, \ldots, \gamma_{h+1}$ such that:

$$\pi_1 \ = < \ldots;\beta;\ldots;\delta_1;\ldots;\gamma_1;\ldots;\delta_h;\ldots;\gamma_h;\ldots;\delta_{h+1};\ldots;\alpha;\ldots >$$

where $L \in Post(\delta_i)$ and $\neg L \in Post(\gamma_i)$. Then: $\beta \prec \delta_1$, $\delta_1 \prec \gamma_1$, $\gamma_1 \prec \delta_1$, $\ldots$, $\delta_{h+1} \prec \alpha$ all belong to $Ab(\pi_1)$. Hence π_2 is not a linearization of $Ab(\pi_1)$. Contradiction. $\square$

Propositions 17 and 18 show that the universe of strict totally ordered plans can be partitioned into equivalence classes, where two totally ordered plans are equivalent if and only if they have the same partially ordered abstraction. Each equivalence class therefore corresponds to a partially ordered plan that can be represented by a set of ordering constraints. Moreover, each totally ordered plan in the class can simply be obtained as a linearization of the partially ordered plan (i.e., the set of ordering constraints). Inasmuch as there exist many linearizations of the same partially ordered plan, the average search complexity of the problem can be greatly reduced if the search for solutions is performed in the space of partially ordered plans rather than in that of totally ordered plans. A logical model for the partial order planning problem and improved search strategy is discussed in the next section.

5.2 – Systematic Search

In searching for solutions to a planning problem, alternative linearizations of the same partially ordered plan Ap need not be generated. This condition, often referred to as *systematicity* [22], leads to more efficient searching as it reduces the search space.

A systematic search can be implemented by defining a notion of *canonical* linearization of a partially ordered plan, and then constraining the search algorithm so as to avoid the generation of non-canonical totally ordered plans. Since the universe of strict totally ordered plans can be partitioned into equivalence classes — each represented by a different partially ordered plan — searching in the space of canonical plans (rather than in that of unrestricted totally ordered plans) does not sacrifice the completeness of the search.

As already pointed out in the previous section, a partially ordered plan A_P can be represented by an acyclic graph G where each action is represented by a node, where there is a directed arc from node α to node β if and only if $\alpha \prec \beta \in A_P$, and where the node representing the initial action `begin` is the origin of G.

We now introduce the notion of *layer* of an action in a partially ordered plan A_P. The layer of an action α in A_P is the maximal distance (i.e., the length of the longest path) from the origin to α in the acyclic graph representing A_P. We shall denote the layer of an action α by $Layer(\alpha)$. Intuitively speaking, the layers of a partially ordered plan A_P define a *greedy* stratification of the plan into sets of independent actions. Namely, if $Layer(\alpha) = Layer(\beta)$ then the α and β neither cause or inhibit one another. The stratification is greedy in the sense that if $Layer(\alpha) = L$ then L is the *first* layer to which α can belong. Indeed, by definition of layer, if $Layer(\alpha) = L$ then there exist $\beta_1, \ldots, \beta_{L-1}$ such that:

- $layer(\beta_j) = L_j \ \forall j \in [1, L-1]$, and
- $\beta_L \prec \alpha \in A_P$, and
- $\beta_j \prec \beta_{j+1} \in A_P \ \forall j \in [1, L-2]$.

We now introduce the notion of *canonical plan* that is based on the notion of layer of an action.

Definition 19
Let π be a linearization of a partially ordered plan Ap. Then π is called canonical *if and only if for every pair of actions in π, say α_j and α_{j+k}:*

(1) $Layer(\alpha_j) \leq Layer(\alpha_{j+k})$, and

(2) If $Layer(\alpha_j) = Layer(\alpha_{j+k})$ then α_j precedes α_{j+k} in lexicographical order (or in some other arbitrarily pre-defined total order between actions).

It is easy to show that once a lexicographical order is chosen, every partially ordered plan Ap has exactly one canonical linearization.

We now present $\mathrm{Datalog}_{1S}$ rules for constructing a canonical plan. The basic idea of the search algorithm is to incrementally construct a canonical totally ordered plan. After inserting n actions, the current plan will have the form

$$\pi \ = < \mathtt{begin}; \alpha_1; \ldots; \alpha_{n-1} >$$

where $Layer(\alpha_{n-1}) = L$. Let now β be an action that is firable in the state resulting from the execution of π. β is added to the current plan π only if either:

(a) β depends on some action α_h in π such that $Layer(\alpha_h) = L$. That is, β is caused by or inhibits some α_h in π such that $Layer(\alpha_h) = L$, or

(b)
- β does not depend on any action α_h in π such that $Layer(\alpha_h) = L$, and
- β depends on some action α_k in π such that $Layer(\alpha_k) = L-1$, and
- β lexicographically follows all actions α_h in π such that $Layer(\alpha_h) = L$.

It is easy to see that every plan found by the above algorithm is a canonical plan. Moreover, the above search algorithm finds all possible canonical plans. Indeed a firable action β is not added to the current partial plan

$$\pi \ = < \mathtt{begin}; \alpha_1; \ldots; \alpha_{n-1} >$$

only if β does not depend on any action α_h in π such that $Layer(\alpha_h) = L$, and
- either β does not depend on any action α_k in π such that $Layer(\alpha_k) = L-1$,
- or β depends on some action α_k in π such that $Layer(\alpha_k) = L-1$, but β does not lexicographically follow all actions in π with layer L. It is easy to see that in either case the plan

$$\pi' \ = < \mathtt{begin}; \alpha_1; \ldots; \alpha_{n-1}; \beta >$$

is not canonical. (In the first case because $Layer(\beta) < L$ by strictness — in the second case by the lexicographical ordering property.)

It is important to notice that the search algorithm only needs to check actions in the current plan that belong to the last two strata considered (the current stratum L and the previous stratum $L-1$). We hence introduce two predicates `last_strat` and `curr_strat`, which hold the actions in the previous stratum and in the current stratum, respectively. These predicates are then used to decide whether the chosen action will be part of the next stratum or of the current stratum.

We begin by pre-computing the `prec` relation that defines the cause/inhibit dependencies between actions. The causality relation between two actions α and β can be defined as follows:

$$\mathtt{prec}(\alpha, \beta) \ \leftarrow \ \mathtt{postcond}(\alpha, \mathtt{X}, \mathtt{Type}), \mathtt{precond}(\beta, \mathtt{X}, \mathtt{Type}). \qquad (14)$$

where the relation `precond` is defined analogously to `postcond` (Section 3). The above rule establishes that action α establishes some condition `X` needed by β for firing. Before a plan is actually constructed, this rule only establishes a pattern of possible dependencies, where for each condition `X` there might be several producers. However, once a plan is actually constructed, the strictness condition ensures that there is only one producer for `X`. Thus we will evaluate these rules dynamically.

If we now define:

$$\texttt{opposite(pos,neg)}. \tag{15}$$

$$\texttt{opposite(neg,pos)}. \tag{16}$$

then the precedence corresponding to the "inhibit" condition can be defined as follows:

$$\begin{aligned}\texttt{prec}(\alpha,\beta) \;\leftarrow\; & \texttt{precond}(\alpha,\texttt{X},\texttt{Type}),\\ & \texttt{postcond}(\beta,\texttt{X},\texttt{UnType}),\\ & \texttt{opposite}(\texttt{Type},\texttt{Untype}).\end{aligned} \tag{17}$$

The rules defining the `firable` actions remain the same as for the totally ordered plans. However, the old rule for `fired` (3) specifying that a choice can be made at random from the rules currently firable must now be refined so as to consider only actions that can occur either in the next layer or in the current one in a canonical linearization.

Therefore, a firable action can be added to the plan as part of the next layer as follows:

$$\begin{aligned}\texttt{next}^{\texttt{J}}(\beta) \;\leftarrow\; & \texttt{firable}^{\texttt{J}}(\beta),\\ & \texttt{curr\_strat}^{\texttt{J}}(\alpha),\\ & \texttt{prec}(\alpha,\beta).\end{aligned} \tag{18}$$

Actions that can be assigned to the current layer can be expressed as follows:

$$\begin{aligned}\texttt{same}^{\texttt{J+1}}(\beta) \;\leftarrow\; & \texttt{firable}^{\texttt{J+1}}(\beta),\neg\texttt{next}^{\texttt{J+1}}(\beta),\\ & \texttt{last\_strat}^{\texttt{J+1}}(\alpha),\texttt{prec}(\alpha,\beta),\\ & \texttt{fired}^{\texttt{J}}(\gamma),\texttt{follows}(\beta,\gamma).\end{aligned} \tag{19}$$

In other words, an action β is assigned to the current layer if:

(i) β need not be in next layer,

(ii) there is a direct dependency from some action α in the previous layer and β, and

(iii) β follows the last fired action γ ($\mathtt{follows}(\beta, \gamma)$), according to a lexicographical order.

Finally, the predicates `curr_strat` and `last_strat` are defined by the following rules:

$$\mathtt{curr\_strat}^{\mathtt{J+1}}(\alpha) \;\leftarrow\; \mathtt{curr\_strat}^{\mathtt{J}}(\alpha), \mathtt{fired}^{\mathtt{J}}(\beta), \mathtt{same}^{\mathtt{J}}(\beta). \tag{20}$$

$$\mathtt{curr\_strat}^{\mathtt{J+1}}(\alpha) \;\leftarrow\; \mathtt{fired}^{\mathtt{J}}(\alpha). \tag{21}$$

$$\mathtt{last\_strat}^{\mathtt{J+1}}(\alpha) \;\leftarrow\; \mathtt{last\_strat}^{\mathtt{J}}(\alpha), \mathtt{fired}^{\mathtt{J}}(\beta), \mathtt{same}^{\mathtt{J}}(\beta). \tag{22}$$

$$\mathtt{last\_strat}^{\mathtt{J+1}}(\alpha) \;\leftarrow\; \mathtt{curr\_strat}^{\mathtt{J}}(\alpha), \mathtt{fired}^{\mathtt{J}}(\beta), \mathtt{next}^{\mathtt{J}}(\beta). \tag{23}$$

It is worth observing that, in terms of XY-stratification, the heads of the rules are always interpreted as 'new' values. Thus $\mathtt{fired}^{\mathtt{J}}(\gamma)$ in rule (19) refers to an old value, while $\mathtt{next}^{\mathtt{J}}(\beta)$ and $\mathtt{same}^{\mathtt{J+1}}(\beta)$ in the rules (18) and (19) refer to new values. Rules (18) and (19) can now be combined to define the new subset of firable actions as follows:

$$\mathtt{subfirable}^{\mathtt{J}}(\beta) \;\leftarrow\; \mathtt{same}^{\mathtt{J}}(\beta). \tag{24}$$

$$\mathtt{subfirable}^{\mathtt{J}}(\beta) \;\leftarrow\; \mathtt{next}^{\mathtt{J}}(\beta). \tag{25}$$

Actions that do not depend on either the current stratum or the last one are not part of the `subfirable` set. Thus our new firing rule is:

$$\mathtt{fired}^{\mathtt{J}}(\alpha) \;\leftarrow\; \mathtt{subfirable}^{\mathtt{J}}(\alpha), \mathtt{choice}^{\mathtt{J}}(\alpha). \tag{26}$$

together with the rule for stage `0` (where no check must be performed):

$$\mathtt{fired}^{\mathtt{0}}(\alpha) \;\leftarrow\; \mathtt{firable}^{\mathtt{0}}(\alpha), \mathtt{choice}^{\mathtt{0}}(\alpha). \tag{27}$$

Example 4 (Blocks world revised)
Consider again the blocks world example. We now extend the description given in Section 2.1 in order to model a multi-agent situation, in which several robots may operate in the environment. Namely the description of actions is extended with the name of the robot. The blocks world domain then contains the following predicates:

`hand_empty(R)`:	*The hand of robot* `R` *is empty*
`holding(R,X)`:	*Robot* `R` *is holding block* `X` *in its hand*
`on(X,Y)`:	*Block* `X` *is on block* `Y`
`ontable(X)`:	*Block* `X` *is on the table*
`clear(X)`:	*There is nothing on top of block* `X`

The following (strict) actions are used:

`pickup(R,X)`: *Robot* `R` *picks up block* `X` *from the table.*

```
Pre  = {ontable(X), clear(X), hand_empty(R), ¬holding(R,X) }
Post = {¬ontable(X), ¬clear(X), ¬hand_empty(R), holding(R,X)}
```

`putdown(R,X)`: *Robot* `R` *puts block* `X` *on the table.*

```
Pre  = {holding(R,X), ¬ontable(X), ¬clear(X), ¬hand_empty(R) }
Post = {¬holding(R,X), ontable(X), clear(X), hand_empty(R) }
```

`stack(R,X,Y)`: *Robot* `R` *puts block* `X` *on top of block* `Y`.

```
Pre  = {clear(Y), holding(R,X), ¬on(X,Y), ¬clear(Y), ¬hand_empty(R) }
Post = {¬clear(Y), ¬holding(R,X), clear(Y), on(X,Y), hand_empty(R)}
```

`unstack(R,X,Y)`: *Robot* `R` *removes block* `X` *from top of block* `Y`.

```
Pre  = {clear(X), on(X,Y), hand_empty(R), ¬clear(Y), ¬holding(R,X)}
Post = {¬clear(X), ¬on(X,Y), ¬hand_empty(R), clear(Y), holding(R,X)}
```

Consider the problem discussed in Section 2.1, with two robots (`r1` *and* `r2`*, say) operating in the environment. In the initial state there are three blocks* `a`, `b`, *and* `c`. *Blocks* `a` *and* `b` *are on the table, while block* `c` *is on top of* `a`. *The goal is to build a stack of blocks such that block* `a` *is on top of* `b` *which in turn is on top of block* `c`.

Actions `begin` *and* `end` *are now defined as follows:*

```
begin:
Pre  = {start}
Post = {¬start, hand_empty(r1), hand_empty(r2), clear(c), on(c,a),
        ontable(a), clear(b), ontable(b)}
```

```
end:
Pre  = {¬done, ontable(c), on(a,b), on(b,c)}
Post = {done}
```

The shortest solutions (i.e., containing the smallest number of actions) for the above described planning problem Π *contain six actions. For instance the totally ordered plan:*

$$\pi_1 \;=\; <\mathtt{unstack(r1,c,a)};\mathtt{putdown(r1,c)};\mathtt{pickup(r2,b)};\\ \mathtt{pickup(r1,a)};\mathtt{stack(r2,b,c)};\mathtt{stack(r1,a,b)}>$$

is a solution for Π *(for the sake of simplicity* `begin` *and* `end` *are omitted). The partially-ordered abstraction* $Ab(\pi_1)$ *consists of the following ordering constraints:*

$$
\begin{array}{rcl}
\texttt{pickup(r2,b)} & \prec & \texttt{stack(r2,b,c)} \\
\texttt{pickup(r2,b)} & \prec & \texttt{stack(r1,a,b)} \\
\texttt{unstack(r1,c,a)} & \prec & \texttt{putdown(r1,c)} \\
\texttt{unstack(r1,c,a)} & \prec & \texttt{stack(r2,b,c)} \\
\texttt{unstack(r1,c,a)} & \prec & \texttt{pickup(r1,a)} \\
\texttt{unstack(r1,c,a)} & \prec & \texttt{stack(r1,a,b)} \\
\texttt{putdown(r1,c)} & \prec & \texttt{stack(r2,b,c)} \\
\texttt{putdown(r1,c)} & \prec & \texttt{pickup(r1,a)} \\
\texttt{putdown(r1,c)} & \prec & \texttt{stack(r1,a,b)} \\
\texttt{stack(r2,b,c)} & \prec & \texttt{stack(r1,a,b)} \\
\texttt{pickup(r1,a)} & \prec & \texttt{stack(r1,a,b)}
\end{array}
$$

The linearizations of $Ab(\pi_1)$ *are (besides* π_1 *itself):*

$$
\begin{array}{rcl}
\pi_2 & = & < \texttt{pickup(r2,b); unstack(r1,c,a); putdown(r1,c);} \\
 & & \quad \texttt{stack(r2,b,c); pickup(r1,a); stack(r1,a,b)} > \\
\pi_3 & = & < \texttt{unstack(r1,c,a); pickup(r2,b); putdown(r1,c);} \\
 & & \quad \texttt{stack(r2,b,c); pickup(r1,a); stack(r1,a,b)} > \\
\pi_4 & = & < \texttt{unstack(r1,c,a); pickup(r2,b); putdown(r1,c);} \\
 & & \quad \texttt{pickup(r1,a); stack(r2,b,c); stack(r1,a,b)} > \\
\pi_5 & = & < \texttt{unstack(r1,c,a); putdown(r1,c); pickup(r2,b);} \\
 & & \quad \texttt{stack(r2,b,c); pickup(r1,a); stack(r1,a,b)} > \\
\pi_6 & = & < \texttt{pickup(r2,b); unstack(r1,c,a); putdown(r1,c);} \\
 & & \quad \texttt{pickup(r1,a); stack(r2,b,c); stack(r1,a,b)} > \\
\pi_7 & = & < \texttt{unstack(r1,c,a); putdown(r1,c); pickup(r1,a);} \\
 & & \quad \texttt{pickup(r2,b); stack(r2,b,c); stack(r1,a,b)} >
\end{array}
$$

Notice that the only canonical plan among all these plans is π_6*.*

5.3 – Properties of Systematic Search Programs

It important to observe that the modelization of partially ordered plans and systematic search satisfies the properties presented in Section 4 for the case of totally ordered plans.

As shown in the previous sections, the systematic search of a canonical solution for a planning problem $\Pi =< S, \mathcal{A}, G >$ can be modeled by translating Π into a choice logic program $\mathcal{CH}(\Pi)$ by:

(i) Associating a clause (1) with each action $\alpha \in \mathcal{A}$, a clause (5) with each action postcondition, and an analogous clause with each action precondition,

(ii) Associating clauses (8) and (9) with each predicate `p`, and

(iii) Including verbatim clauses (2), (4), (6), (7), and (14) — (27).

Such a choice logic program $\mathcal{CH}(\Pi)$ can be converted into a logic program with negation $foe(\mathcal{CH}(\Pi))$, by transforming clauses (26) and (27) as discussed in Section 4. It is easy to observe that the program $\mathcal{CH}(\Pi)$ modeling the systematic search in the space of partially ordered plans is locally stratified modulo choice (i.e., Proposition 2 continues to hold). Therefore the iterated choice fixpoint procedure can be employed to compute the stable model semantics of $foe(\mathcal{CH}(\Pi))$ (Theorems 5 and 6).

The properties of the stable models of the program $foe(\mathcal{CH}(\Pi))$, presented in Section 4.1 for the case of totally ordered plans, continue to hold also in the case of partially ordered plans. More precisely, Propositions 7 and 8 continue to hold. (In the formulation of the latter, `firable` is obviously replaced by `subfirable`.)

Proposition 9 states the correspondence between the existence of a solution for a planning problem Π and the existence of a stable model for the corresponding logic program $foe(\mathcal{CH}(\Pi))$. An analogous result holds for the case of partially ordered plans.

Proposition 20
Let Π *be a planning problem.*

π *is a totally ordered solution for* Π

$\Longleftrightarrow$

$\exists$ *a stable model* M *of* $foe(\mathcal{CH}(\Pi))$ *s.t.:*

(i) $\{\mathtt{fired}^0(\mathtt{begin}), \mathtt{fired}^1((\alpha_1)\vartheta_1), ..., \mathtt{fired}^{\mathtt{n}}((\alpha_{\mathtt{n}})\vartheta_{\mathtt{n}}), \mathtt{fired}^{\mathtt{n+1}}((\mathtt{end})\vartheta_{\mathtt{n+1}})\} \subseteq M$, *and*

(ii) π *is a linearization of* $Ab(<(\alpha_1)\vartheta_1, ..., (\alpha_n)\vartheta_n>)$.

The above result descends from the facts that the universe of strict totally ordered plans can be partitioned into equivalence classes (Proposition 17 and 18), and that the systematic search strategy always finds exactly one representative for each equivalence class (viz., the canonical plan). Therefore each stable model for the program $foe(\mathcal{CH}(\Pi))$ defines a canonical solution for Π inasmuch as the clauses given in Section 5.2 directly translate the conditions of Definition 19.

It is also important to observe that all the properties presented in Section 4.2 concerning the amenability to efficient implementation hold also for the formalization of partially ordered plans. In particular the primed dependency graph of the program is acyclic, as illustrated in Figure 3. Therefore the program is XY-stratified and the properties of Lemma 11, stating the existence of stable models and their polynomial-time computability, continue to hold.

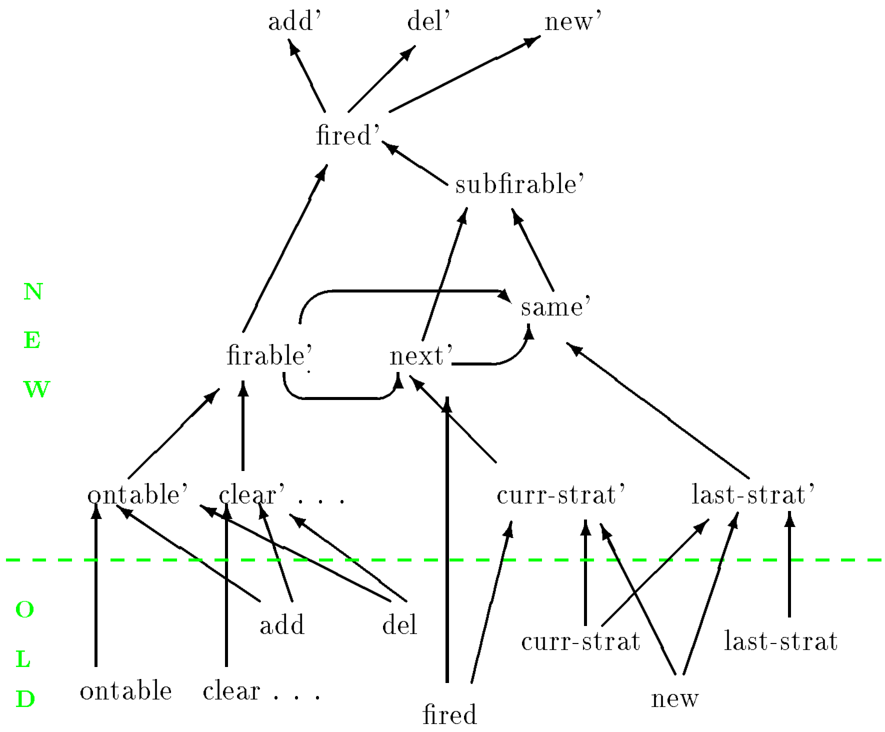

Figure 3: Primed dependency graph for the blocks world example (partially ordered plans).

6 – Related Work

Over the last few years, there has been a substantial amount of work linking logic and planning (e.g., [12, 15, 17, 23, 24, 25]). Of all these efforts, the paradigm integrating planning and logic that stands out is the situation calculus proposal by McCarthy [23] — this proposal has been extended by Reiter and his colleagues[25]. Almost all other works in this area build upon the situation calculus. In this section, we will provide a brief comparison between our work and situation calculus based efforts. Our framework is similar to the situation calculus in many ways. Some of these similarities, as well as differences are listed below.

Key Similarity
First, our framework, like the situation calculus, expands every predicate symbol by adding a new argument that captures the state. Thus, for instance, a state term in the situation calculus may look like:

$$\mathtt{do(go(place3, place2, (go(place3, place4, s_0))))}.$$

This denotes the state reached by executing first the action "Go from place 3 to place 4" in the initial state (s_0), and then "Go from place 3 to place 2". The final state reached is denoted by the above state term. Thus, in the situation calculus, all states (including unreachable states) have an associated term that can occupy a state-argument slot in (expanded) predicates.

Key Differences

1. In contrast, in our framework, we use *integers* to denote states. This has several advantages. First, not all states are explicitly encoded in the language, thus leading to a smaller number of states. Second, only some reachable states (not all) can actually be generated in our framework. For example, the only way for $p(2, \ldots)$ to be true is to have successfully executed two actions that lead to this state.

2. In our framework, every time a state is reached, one of a number of possible actions to execute is *chosen*. In contrast, in the situation calculus approach, each executable action may be executed to lead to a set of possible states, one of which is subsequently chosen for further expansion.

3. Thus, in our framework, given a goal G that we wish to achieve, the planning problem boils down to checking if there exists a choice model in which a corresponding goal $G^\star$ is true. In contrast, in the situation calculus, an attempt is made to *prove* G using standard theorem proving techniques. Though elegant, this latter approach may suffer from the inefficiencies of theorem proving techniques. In contrast, our approach can take full advantage of efficient optimizations in deductive databases, such as effective use of XY-stratification.

4. Furthermore, as shown in [29], our framework may be easily extended to incorporate updates (or "surprises") that occur during the execution of the plan. Despite years of work in the situation calculus area, there had really been no success in this arena.

5. Finally, our basic approach can be extended to model more complex planning strategies. In the second part of the paper, in particular, we have shown how to support partially ordered plans and optimization using systematicity.

Several works have addressed the problem of modeling the logic of actions by means of logic programming languages. Eshgi and Kowalski [13], and Evans [14] showed how negation by failure can be successfully employed to correctly formalize the so-called *Yale shooting problem* [18].

Apt and Bezem [2] further elaborated that the Yale Shooting Problem (YSP), as well as some other temporal reasoning problems, can be modeled by means of a special class of logic programs with negation, called *acyclic* programs. Acyclic programs enjoy several interesting properties: For instance, an acyclic program P has a unique Herbrand model M_P, which is also the unique perfect model of P and the unique fixpoint of the immediate consequence operator T_P. An acyclic program P can be queried by means of SLDNF [19] to refute non-floundering formulae in M_P. For instance, the acyclic YSP program can be queried by means of SLDNF via a query of the kind $\leftarrow holds(dead, [shoot, wait, head])$ to determine whether the fluent *dead* holds in the (given) situation $[shoot, wait, head]$. On the other hand, it is worth observing that acyclic programs may not allow one to construct — via a top-down SLDNF computation — a sequence of events after which a given fluent will hold. For instance, the query $\leftarrow holds(alive, L)$ may flounder on program YSP.

As shown in [2], when used for coding temporal reasoning problems, acyclic programs provide a logic programming counter-part of the situation calculus. In this perspective, the previous analysis of the relations between our approach and situation calculus carries over to the relations with acyclic programs. As in the case of situation calculus, Apt and Bezem do not address the issue of how to *choose* an action out of a set of executable actions. For instance, the bottom-up computation of the YSP program constructs a single model which describes all possible states yielded by all possible sequences of events. In our approach, instead, each bottom-up computation actually *chooses* (and finds) a sequence of actions and each stable model of our XY-programs denotes a single possible sequence. Computationally, the problem of checking whether a given sequence of actions is a valid plan is a a much simpler problem than finding a valid plan — inasmuch as checking if a model is stable is a polynomial problem while finding a stable model is exponential. In our framework this simpler problem can be formulated by observing that a candidate plan is represented in the form:

$$\mathcal{S} =< \texttt{fired}^0(\texttt{begin}); \texttt{fired}^1(\alpha_1); ...; \texttt{fired}^{\mathrm{n}}(\alpha_{\mathrm{n}}); \texttt{fired}^{\mathrm{n+1}}(\texttt{end}) > \tag{28}$$

Thus the problem reduces to checking whether such a plan can be generated by the $\mathrm{Datalog}_{1S}$ rules for the planning problem at hand. By a simple inspection of rule (10), we see that the sequence $\mathcal{S}$ of atoms in equation (28) can be generated by our $\mathrm{Datalog}_{1S}$ program if and only if the following choices are made at each step in the computation:

$$\mathcal{C} =< \texttt{chosen}^0(\texttt{begin}); \texttt{chosen}^1(\alpha_1); ...; \texttt{chosen}^{\mathrm{n}}(\alpha_{\mathrm{n}}); \texttt{chosen}^{\mathrm{n+1}}(\texttt{end}) > \tag{29}$$

Therefore, in order to check whether the sequence $\mathcal{S}$ (equation (28)) is a valid plan, we can add the atoms $\mathcal{C}$ (equation (29)) as unit clauses to our program. The inclusion of these facts in the initial sets prevents the generation of any other $\texttt{chosen}^{\mathrm{j}}$ fact, for $0 \leq j \leq n+1$ — rules (11) e (12) could in fact be deleted. This ensures that the computation is deterministic and thus polynomial until step $n+1$ (where it terminates given the special properties of the *end* action). If rule (10) fails, for any $0 \leq j \leq n+1$ (e.g., because the action chosen is not among those firable at that step), then the plan is not valid. On the other hand, if the computation reaches step $n+1$ then the plan has been proven valid. The computation so constrained by the introduction of the `chosen` facts, is quite efficient inasmuch as it reduces to applying the frame axioms and checking whether the chosen rule is in fact firable.

Our approach to modeling the logic of planning also relates to other uses of logic for knowledge representation, such as meta-logics (e.g., [3, 4, 6]). Indeed the representation of a planning problem Π by means of a logic program discussed in Section 3 bears strong similarities with a representation of Π by means of meta-programming. The process of planning can be described by means of a meta-interpreter that simulates the execution of a set of actions $\mathcal{A}$, suitably represented as an object program. In this view, clauses (1), (5), (8) and (9) of Section 3 can be viewed as a representation of the object program coding a planning problem Π, while clauses (2), (3), (4), (6) and (7) as the definition of the meta-interpreter. Actually, a natural meta-logical definition of the planning process would represent world states as terms, and the execution of actions as a transformation of such terms. The $\mathrm{Datalog}_{1S}$ representation of a planning problem Π may be viewed as a compiled version of such a meta-logical representation of Π, where stage arguments of $\mathrm{Datalog}_{1S}$ predicates are used to code dynamic changes of the state. The $\mathrm{Datalog}_{1S}$ representation significantly improves the efficiency of a meta-interpretative representation (which suffers from the interpretation overhead), while still being general and flexible.

7 – Conclusions

In this paper, we have shown that complex planning problems and non-monotonic strategies find natural logic-based formulations in the framework of deductive database languages. In particular, we have shown how totally and partially ordered plans can both be modeled using XY-stratified Datalog_{1S} programs. The framework proposed is flexible, amenable to efficient implementation, and supportive of variations and improvements. For instance, notions such as rigid plans and surprises can also be included in the model [29]. In [5, 7], we generalize our logic-based framework to handle parallel plans, that is, plans where actions that are compatible under some concurrency model are executed in parallel.

Since the Datalog-oriented approach discussed in this paper represents a minimalist approach to the complex problems occurring in the planning domain, there is a need to investigate and understand its limits in modeling other complex planning problems. In particular, several new concepts and techniques have recently emerged in the field of planning research. For instance, in *least-commitment* planning, search is performed in the space of partially ordered plans, rather than in the space of world states (see [33] for a good discussion of the method and its benefits). Current work is oriented in this direction.

Another interesting extension is to consider the problems of temporal projection with incomplete information, as well as reasoning about the past. Gelfond and Lifschitz [17] addressed these issues by representing properties of actions in a logic programming language that uses both classical negation and negation by failure. Future work will be devoted to investigate the extendibility of our approach to such more complex forms of temporal reasoning.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments and suggestions. The first author has been partly supported by C.N.R. grant 203.01.62. The second author has been partly supported by ARO grants DAAL-03-92-G-0225 and DAAH-04-95-10174, by NSF grants IRI-91-09755 and IRI-93-57756, by AFOSR grant F49620-93-1-0065, and by ARPA/Rome Labs contract Nr. F30602-93-C-0241 (Order Nr. A716). The third author has been partly supported by NSF grant IRI-93-14905.

References

[1] S. Abieteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.
[2] K.R. Apt and M. Bezem. Acyclic programs. In D.H.D. Warren and P. Szeredi, editors, *Proceedings Seventh International Conference on Logic Programming*, pages 617–633. The MIT Press, 1990.
[3] K.A. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, 3(3):359–383, 1985.
[4] K.A. Bowen and R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.
[5] A. Brogi, V.S. Subrahmanian, and C. Zaniolo. Modeling sequential and parallel plans. Technical report, Submitted for publication, 1996.
[6] A. Brogi and F. Turini. Metalogic for Knowledge Representation. In J.A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pages 100–106. Morgan Kaufmann, 1990.
[7] A. Brogi and C. Zaniolo. Converting linear and nonlinear plans into parallel execution plans. Technical report, Department of Computer Science, UCLA, 1995.
[8] J. Chomicki. Polynomial-time computable queries in temporal deductive databases. In *PODS'90*, 1990.
[9] J. Chomicki. Temporal deductive databases. In A. Tansel, J. Clifford, S. Gadia, S. Jagodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, pages 294–320. Benjamin/Cummings, 1993.
[10] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, 1978.
[11] L. Corciulo, F. Giannotti, and D. Pedreschi. Datalog with non-deterministic choice compute ndb-ptime. In *Procs., International Conference on Deductive and Object-Oriented Databases, DOOD'93*, 1993.
[12] K. Erol, D.S. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 1995.
[13] K. Eshgi and R.A. Kowalski. Abduction compared with negation as failure. In G. Levi and M. Martelli, editors, *Proceedings Sixth International Conference on Logic Programming*, pages 234–254. The MIT Press, 1989.
[14] C. Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proceedings Second Int'l Symp. on Artificial Intelligence*, 1989.
[15] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving in problem solving. *Artificial Intelligence*, 2(3,4):189–208, 1971.
[16] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K. Bowen, editors, *Logic programming: Proceedings of the fifth Int'l Conf. and Symposium*, pages 1070–1080, 1988.
[17] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic programming*, 17:301–322, 1993.
[18] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.
[19] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, second edition, 1987.
[20] V. Marek and V.S. Subrahmanian. The relationship between stable, supported, default and auto-epistemic semantics for general logic programs. *Theoretical Computer Science*, 103:365–386, 1992.
[21] V. W. Marek and M. Truszczynski. *Nonmonotonic Logic*. Springer-Verlag, 1995.
[22] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639. The MIT Press, 1991.

[23] J. McCarthy. Applications of circumsription to formalising common sense reasoning. *Artificial Intelligence*, 26:89–116, 1986.

[24] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[25] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Proc. 1993 Intl. Conf. on Logic Programming*, pages 203–221. MIT Press, 1993.

[26] T. Przymusinski. On the declarative and procedural semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[27] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. 9th ACM Symp. on Principles of Database Systems*, 1990.

[28] E. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-75)*, pages 206–214, 1975.

[29] V.S. Subrahmanian and C. Zaniolo. Relating stable models and ai planning domains. In *Procs., 12th International Conference on Logic Programming, ICLP'95*, 1995.

[30] A. Tate. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 889–900, 1977.

[31] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *VLDB Journal*, 3:245–288, 1994.

[32] D.H.D. Warren. An abstract prolog instruction set. Technical report, SRI International, 1983.

[33] D.S. Weld. An introduction to least commitment planning. *A.I. Magazine*, 1995.

[34] C. Zaniolo. A unified semantics for active and deductive databases. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, pages 271–287, 1993.

[35] C. Zaniolo. Transaction-conscious stable model semantics for active database rules. In *Procs., International Conference on Deductive Object-Oriented Databases, DOOD'95*, 1995.

[36] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the ldl++ approach. In *Procs., International Conference on Deductive and Object-Oriented Databases, DOOD'93*, 1993.