A Deductive Database Approach to A.I. Planning

Antonio Brogi* V.S. Subrahmanian †Carlo Zaniolo‡

Abstract

In this paper, we show that the classical A.I. planning problem can be modelled using simple database constructs with logic-based semantics. The approach is similar to that used to model updates and nondetermism in active database rules. We begin by expressing plans by means of Datalog_{1S} programs and nondeterministic choice constructs, for which we provide a formal semantics using the concept of stable models. The resulting programs are characterized by a syntactic structure (XY-stratification) that makes them amenable to efficient implementation using compilation and fixpoint computation techniques developed for deductive database systems. We first develop the approach for sequential plans, and then we illustrate its flexibility and expressiveness by formalizing a model for parallel plans, where several actions can be executed simultaneously. The characterization of parallel plans as partially ordered plans also allows us to reduce the search space and hence to improve the efficiency of the planning process.

---

*Dipartimento di Informatica, Università di Pisa Corso Italia 40, 56125 Pisa, Italy— `brogi@di.unipi.it`

†Computer Science Department, University of Maryland College Park, MD 20742, U.S.A.— `vs@cs.umd.edu`

‡Computer Science Department, University of California, Los Angeles CA 90024, U.S.A.— `zaniolo@cs.ucla.edu`

Contents

1 Introduction

Relational databases have shown the practical feasibility of efficiently supporting a declarative query language with logic-based semantics. Seeking to extend relational query languages to achieve Turing completeness, Deductive Databases have recently introduced constructs that combine usability and amenability to efficient implementation with a formal logic based semantics. In particular, simple constructs have been introduced to effectively express (i) database states and transitions between states and (ii) nondeterminism [31]. These have used to solve an array of database problems ranging from active database rules to user-defined aggregates query languages [31, 34]. But the generality of these extensions and their applications outside database area is not well-understood to date. This paper addresses this issue by showing how these query-oriented concepts can solve the classical A.I. problem effectively—thus contributing to the integration of databases and A.I..

Planning has been one of he first fields of computing to witness the use of logic as a formal model of computation. Early work 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 used to construct plans.

In fact, the area of non-monotonic reasoning has seen a substantial body of work and significant progress since McCarthy's original introduction of the concept of circumscription [23]. In particular, the introduction of the concept of stable models [10] has captured in a simple definition many of the key ideas underlying different approaches proposed for non-monotonic reasoning [21].

While stable model semantics is not without drawbacks (e.g., computational intractability [11], and lack of total stable models for certain programs), recent work on deductive databases has identified classes of programs whose syntactic structure ensures the existence of total stable models and their efficient computability [7, 35, 37].

In this paper, we begin by modelling classical STRIPS-like totally ordered plans using Datalog_{1S} rules [5], and develop an approach that is quite flexible and can be used to model various planning strategies. 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, the data-complexity involved in constructing a stable model is polynomial[1].

(2) The proposed approach is quite expressive and flexible. We provide evidence of this by modelling parallel plans. We show how totally ordered and partially ordered plans can be converted into equivalent parallel plans. Partially ordered plans are abstract plans where 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.

---

[1]In data-complexity analyses, it is typically assumed that there is an a priori bound b on the arity of all predicates involved.

The organization of the paper is as follows. The basic notion of a totally ordered (or sequential) plan is 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 [35]. In Section 5, we show that the proposed model can be easily extended to characterize parallel plans, as well as systematic searches of plans.

2 Preliminaries

This section briefly introduces the concept 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 complete 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 [9], 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 $\{X_1, .., X_n\}$, and

- $Add(\alpha)$ and $Del(\alpha)$ are finite sets of atoms, whose variables are all in 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 [8]. 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. $S' = (S - (Del(\alpha)\vartheta)) \cup (Add(\alpha)\vartheta)$.

If α is ϑ-*executable* in a state S, *resulting* in state S', we also write $S \xrightarrow{\alpha,\vartheta} S'$.

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: $S \xrightarrow{\alpha_1,\vartheta_1} S_1 \xrightarrow{\alpha_2,\vartheta_2} S_2 \ldots \xrightarrow{\alpha_n,\vartheta_n} S_n$.

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 [9]. The standard blocks world domain contains the following predicates:

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

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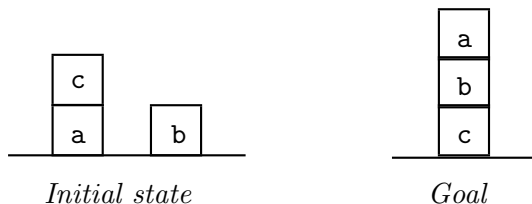`Post = {¬on(X,Y), ¬hand_empty, clear(Y), holding(X)}`

3

Figure 1: Blocks world example.

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, 32], 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   Modelling 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 [7, 27].

We will use Datalog$_{1S}$ to model state changes [5]. The merits of Datalog$_{1S}$ for modelling temporal and dynamic systems have been described, for instance, in [6, 33, 34]. In Datalog$_{1S}$ predicates may have an additional argument called the *stage argument*. We will also use the term *temporal argument* as a synonym of 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 be denoted by $\mathtt{s(J)}$, and this notation is at the root of the name Datalog$_{1S}$. A predicate $\mathtt{p}$ with stage argument J therefore has the form $\mathtt{p(J,\vec{t})}$. For notational convenience, we will write the stage argument as superscript, so that for instance $\mathtt{p(J,\vec{t})}$ will be written as $\mathtt{p^J(\vec{t})}$. The variable J appearing in the stage argument of the head of a rule r will be referred to as the *temporal variable* for r, and also as the stage *stage variable* for r.

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:

$$\mathtt{firable^J}(\alpha) \quad \leftarrow \quad \mathtt{p_1^J(\vec{t}_1), \ldots, p_m^J(\vec{t}_m), \neg q_1^J(\vec{u}_1), \ldots, \neg q_n^J(\vec{u}_n).} \tag{1}$$

where $Pre(\alpha) = \{\mathtt{p_1(\vec{t}_1), \ldots, p_m(\vec{t}_m), \neg q_1(\vec{u}_1), \ldots, \neg q_n(\vec{u}_n)}\}$. 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 $\mathtt{pickup(X)}$). As we shall discuss in Section 4.4, 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 $\mathtt{begin}$ and $\mathtt{end}$, 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 $\mathtt{start}$ is true at stage $\mathtt{0}$, and hence action $\mathtt{begin}$ 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:

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

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 $\mathtt{choice}$ 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 choice goal $\mathtt{choice^J}(\alpha)$ establishes a functional dependency (FD) $\mathtt{J} \to \alpha$, and J will be called the *left side of the choice FD* for rule (3). The second goal in clause (3) ($\neg\mathtt{firable^J(end)}$) enforces

the condition that an action can be fired at stage J only if the special action **end** is not firable at that stage.

Finally:

$$\texttt{fired}^{\texttt{J}}(\texttt{end}) \;\; \leftarrow \;\; \texttt{firable}^{\texttt{J}}(\texttt{end}), \neg\texttt{done}. \tag{4}$$

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

Frame Axioms

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

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

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:

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

The postcondition of **begin** and **end** are:

$(p15)$ $\texttt{postcond}(\texttt{begin}, \texttt{start}, \texttt{neg}).$
$(p16)$ $\texttt{postcond}(\texttt{begin}, \texttt{hand\_empty}, \texttt{pos}).$
$(p17)$ $\texttt{postcond}(\texttt{begin}, \texttt{clear}(\texttt{c}), \texttt{pos}).$
$(p18)$ $\texttt{postcond}(\texttt{begin}, \texttt{on}(\texttt{c}, \texttt{a}), \texttt{pos}).$
$(p19)$ $\texttt{postcond}(\texttt{begin}, \texttt{ontable}(\texttt{a}), \texttt{pos}).$
$(p20)$ $\texttt{postcond}(\texttt{begin}, \texttt{clear}(\texttt{b}), \texttt{pos}).$
$(p21)$ $\texttt{postcond}(\texttt{begin}, \texttt{ontable}(\texttt{b}), \texttt{pos}).$
$(p22)$ $\texttt{postcond}(\texttt{end}, \texttt{done}, \texttt{pos}).$

State changes can be therefore modelled as follows:

$$\texttt{add}^{\texttt{J}}(\texttt{Cond}) \;\; \leftarrow \;\; \texttt{fired}^{\texttt{J}}(\alpha), \texttt{postcond}(\alpha, \texttt{Cond}, \texttt{pos}). \tag{6}$$

$$\text{del}^{\text{J}}(\text{Cond}) \quad \leftarrow \quad \text{fired}^{\text{J}}(\alpha), \text{postcond}(\alpha, \text{Cond}, \text{neg}). \tag{7}$$

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

$$\text{p}^{\text{J+1}}(\vec{\text{t}}) \quad \leftarrow \quad \text{p}^{\text{J}}(\vec{\text{t}}), \neg\text{del}^{\text{J}}(\text{p}(\vec{\text{t}})). \tag{9}$$

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:*

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

The frame axioms lead to the following clauses:

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

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

Our planning programs include nonstratified negation with the nondeterministic construct *choice*. In general, this combination of nonmonotonic and nondeterministic constructs can result in programs that do not have clear semantics, or are semantically well-formed but computationally intractable [28]. Fortunately, planning programs always have a formal semantics based on the concept of total stable models, and an efficient computation based on the concept of XY-stratification (Section 4.3).

4.1 The Choice Construct

The first step consists of eliminating the choice goals by adopting the approach discussed in [27], where a program P with choice goals is converted into a logic program with negation $foe(P)$ that defines its semantics — $foe(P)$ is called the *first-order extension* of P. If P (with the choice goals removed) is a positive program, then $foe(P)$ always has one or more (total) stable models [27, 37]. Each such a model is called a *choice model* for P. We will next see that similar properties hold when P is a program stratified with respect to negation, and when P is an XY-stratified program.

For a program $\mathcal{CH}(\Pi)$ representing a planning problem Π, the choice goal is used in rule (3). This clause is thus expanded into and replaced by the following set of clauses[2]

$$\texttt{fired}^{\mathrm{J}}(\alpha) \;\leftarrow\; \texttt{firable}^{\mathrm{J}}(\alpha), \neg\texttt{firable}^{\mathrm{J}}(\texttt{end}), \qquad (10)$$
$$\texttt{chosen}^{\mathrm{J}}(\alpha).$$

$$\texttt{chosen}^{\mathrm{J}}(\alpha) \;\leftarrow\; \texttt{firable}^{\mathrm{J}}(\alpha), \qquad\qquad (11)$$
$$\neg\texttt{diffchoice}^{\mathrm{J}}(\alpha).$$

$$\texttt{diffchoice}^{\mathrm{J}}(\alpha) \;\leftarrow\; \texttt{chosen}^{\mathrm{J}}(\beta), \alpha \neq \beta. \qquad\qquad (12)$$

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).*

---

[2]When the program contains several choice rules, then we use `chosenr` and `diffchoicer`, where `r` is the unique identifier for the choice rule. This unique identifier is not needed in our planning programs, as there is only one choice rule and no ambiguity can thus occur.

4.2 Topological Stratification

The original planning program $\mathcal{CH}(\Pi)$, without the choice goals, is locally stratified by the first argument (i.e., the temporal) argument. However because of the cycle involving the $\neg\texttt{diffchoice}$ predicate, $foe(\mathcal{CH}(\Pi))$ is not locally stratified. Therefore, we now introduce a new notion, called *topological* stratification, that leads to a layered computation of the stable models of a program.

Let P be a program and Σ_P be a totally ordered partition of B_P, the Herbrand Base of P. Thus, to each stratum in Σ_P there corresponds an integer $0 \leq j < n$ where n is the number of classes in the partition if this is finite, whereas n denotes the limit ordinal ω if the partition contains an infinite number of classes. Now, for each $x \in B_P$, let $stratum(x)$ denote the unique stratum to which x belongs to. When j is a nonnegative integer, we use $ground^j(P)$ to denote the set of all rules in $ground(P)$ whose head belongs to stratum j of Σ_P, while $ground^{<j}(P)$ denotes the set of rules whose head belongs to strata strictly lower than the j-th stratum, i.e., $ground^{<j}(P) = \bigcup_{0 \leq i < j} ground^i(P)$. Also, $ground^{\leq j}(P)$ denotes $ground^{<j}(P) \cup ground^j(P)$. Thus, if Σ_P is a partition of cardinality n, then $ground^{<n}(P) = ground(P)$, where n is either an integer or the symbol ω.

We will say that 'a stratum A is higher (lower) than stratum B' to denote that either A is strictly higher (lower) than B, or that A=B.

Definition 1 Topological Stratification for a program P. *Let Σ_P be a totally ordered partition of the atoms of B_P. Σ_P is called a topological stratification for P, when for every rule $r \in ground(P)$ the head of r belongs to a stratum that is higher than the strata of the goals of r.*

Thus a topological stratification is a relaxation of local stratification, insofar as the negated goals are not required to be in strata strictly lower than the head's stratum; they can also be in the same stratum (i.e., positive goals and negated ones are now treated the same).

The significance of topological stratification follows from the following result[3].

Theorem 1 *Let P be a logic program and Σ_P be a topological stratification for P in n strata. Then, M is a stable model for P iff for every $0 \leq j < n$*

$$M^{\leq j} = \{x \in M | stratum(x) \leq j\}$$

is a stable model for $ground^{\leq j}(P)$.

Theorem 1 relates the stable models for a program to the stable models for its topological strata. The next theorem defines the relation between one stratum and the next one. With M a set of atoms, let $\phi(M)$ denote the set of facts obtained by recasting each atom in M as a fact: $\phi(M) = \{a \leftarrow .|a \in M\}$. Then we have the following theorem.

---

[3]The proof of the theorems for this section are given in Appendix A.

Theorem 2 *Let P be a logic program with a topological stratification Σ_P containing n strata. Then, for every $0 \le j < n$:*

1. *If M is a stable model for $ground^{\le j}(P)$, then every stable model for $\phi(M) \cup ground^{j+1}(P)$ is a stable model for $ground^{\le j+1}(P)$, and*

2. *If N is a stable model for $ground^{\le j+1}(P)$ then $N^{\le j} = \{x \in N | stratum(x) \le j\}$ is a stable model for $ground^{\le j}(P)$.*

Theorem 2 suggests the following iterative procedure to compute the stable model of a program P that is topologically stratified in n strata.

Procedure 1 *Iterated Stable Model computation for a program P with topological stratification Σ_P containing n strata:*

Base: Let M^0 be a stable model for $ground^0(P)$.

Induction: For $j = 0, \ldots, n-1$, let M^{j+1} be a stable model for $\phi(M^j) \cup ground^{j+1}(P)$.

Result: $M^\omega = \bigcup_{0 \le j < n} M^j$

It follows immediateness from Theorem 2 that the iterated stable model computation provides a sound and nondeterministically complete procedure to compute the stable model of a program.

These results can now be applied directly to the computation of stratified choice programs. If P is a choice program, $det(P)$ denotes the program obtained from P by removing its choice goals.

Definition 2 *Let P be a choice program where the rules may contain negated goals. If $det(P)$ is stratified, then P is said to be a* stratified choice program.

As shown in Appendix A, stratification of $det(P)$ induces a topological stratification in $foe(P)$; therefore:

Lemma 3 *Let P be a stratified choice program. Then $foe(P)$ has one or more stable models.*

For stratified choice programs, the iterated stable model computation is basically the same as the iterated fixpoint computation for stratified programs. For strata without choice goals, we compute the least model by the usual ω-power computation; but for strata where some rules contain choice goals we must instead compute a choice model.

Consider now a planning problem $\mathcal{CH}(\Pi)$; this program is topologically stratified by the value of the temporal argument in the head of each rule. Moreover, if we instantiate all the variables in the heads of its rules to yield the same stage value, then we obtain a program that can be viewed as a stratified choice program; thus each stratum, and, therefore, the whole program has one or more stable models. Thus, we can now state the following theorem.

Theorem 4 *Each planning program has one or more stable models.*

4.3 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 5 *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. $\square$

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 $\mathtt{J} - 1$ included, but which causes action α to be fired at stage $\mathtt{J}$.

Proposition 6 *Let Π be a planning problem, and let M be a stable model of $foe(\mathcal{CH}(\Pi))$. Let $[M]_{\mathtt{J}} = \{\mathtt{q}^{\mathtt{I}}(\vec{\mathtt{t}}) \mid \mathtt{q}^{\mathtt{I}}(\vec{\mathtt{t}}) \in M$ 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. $\square$

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 7 *Let Π be a planning problem.*

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

The above proposition shows that any planning problem Π can be converted into a logic program with nonmonotonic 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 8 *Let Π be a planning problem, let $\pi = <(\alpha_1)\vartheta_1; \ldots; (\alpha_n)\vartheta_n>$ be a sequence of actions, and let*

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

Then:

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

4.4 Datalog$_{1S}$ and XY-Stratification

In the previous sections, we have ensured the existence of declarative semantics based on the notion of stable models for our planning problem. In this section, we show that the semantic well-formedness of these programs follows from the syntactic structure of XY-Stratified Datalog$_{1S}$ programs [35]. These programs are amenable to efficient implementation, and actually supported in various deductive database systems [35, 30]. Also, we briefly discuss how to model two variations of the planning problem, one having to do with verifying plans, the second with using heuristics in searching for plans. In the following section, we show that parallel plans and partial order plans can also be described using XY-stratified programs.

Therefore, the syntactic structure studied here can be used to model several planning problems, which thus inherit the desirable semantic and computational properties of XY-stratified programs (e.g., XY-stratification can be checked at compile time, whereas the existence of total well-founded models or stable models cannot [35]).

The idea of XY-stratification can be explained by viewing heads of recursive Datalog$_{1S}$ rules as defining new stage values for the predicates.

Definition 3 *Let P be a set of rules defining mutually recursive predicates. Then, P is an XY-program if each recursive predicate in the body has a temporal argument that is either equal or one less than that in the head. A rule r is said to be an X-rule when all the recursive predicates share the same argument, and a Y-rule otherwise. A program consisting of X-rules and Y-rules is said to be an XY-program.*

For our planning problem, the frame axiom rules are Y-rules, and the other rules are all X-rules. Thus, our planning program is an XY-program. Now there is a simple syntactic check that guarantees the semantic well-formedness of an XY-stratified program P. The test begins by priming the recursive predicates in P to yield the *primed version* of P, denoted P' and constructed as follows:

> For each rule $r \in P$, prime all the recursive predicates in r that have the same temporal argument as the head of r. All other occurrences of recursive predicates remain unprimed.

The *primed dependency graph* for the XY-program P is simply the dependency graph for the P' program so derived.

For the example at hand, for instance, the primed dependency graph is shown in Figure 2. Observe how the primed and unprimed versions of the recursive predicates respectively denote these predicates for the 'new' and 'old' values of temporal arguments.

Definition 4 *Let P be a choice program. Then P is said to be an XY-stratified choice program when the following three conditions hold:*
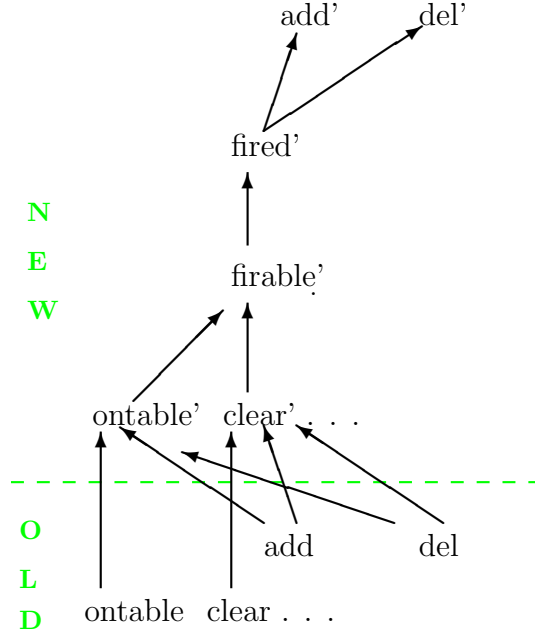
Figure 2: Primed dependency graph for the blocks world example.

- *P is an XY-program,*
- *The primed version of P is a stratified choice program,*
- *If r is a recursive choice rule, then, the left side of each choice FD for r contains r's stage variable.*

XY-stratified choice programs always have one or more stable models[4].

Theorem 9 *Every XY-stratified choice program has one or more stable models*

Procedure 1 is here greatly simplified for an XY-stratified program P, because:

- only the results from the old stratum are here needed to compute the model for the next stratum,
- the rules used in the computation do not change from one stratum to the next, except for the temporal arguments. Therefore, let P_{bis} denote the program obtained from P', the primed version of P, by dropping the temporal arguments. P_{bis} is called the *bistate version* of P. Procedure 1 then reduces to an iteration loop over the computation of the stratified choice program P_{bis}, where predicates no longer have temporal arguments. The temporal argument is in fact computed separately as the loop counter.

---

[4]The proof for this theorem is given in the Appendix A.

13

Thus, P_{bis} is used at compile time to verify that the XY-stratification condition holds, and it is also used at run time to efficiently compute a stable model. Moreover, observe that the primed dependency graph for a planning problem does not contain any strong component. Programs such as this are particularly well-behaved as the iterated fixpoint is no longer transfinite but 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 $J+1$ in the bottom-up order defined by 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 proceed in a similar fashion. Therefore, Algorithm 1 simplifies dramatically for XY-stratified choice programs. For rules (1), (2), (5), (6), (7), and (8) there is a single computation step of T_P.

Finally, observe the rules in Equations 8 and 9, which derive the new state using the old state together with the add and delete lists. We can execute rule (9) first, and observe that, with the temporal argument stored separately, this rules can be implemented by simply deleting the axioms in `del` from the state predicate `p`. Thus, multiple copies of this predicate need not be stored as it can simply be updated in-place. (The $\mathcal{LDL}++$ compiler implements this optimization under the so-called 'copy rule optimization' [35]). Thus, we have achieved a 'best of two worlds' situation: frame axioms are modelled in declarative logic, which is then mapped into efficient imperative insert/delete instructions by the smart compiler.

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)`. But strict Datalog$_{1S}$ does not allow the use of function symbols [6], other than the successor function `+1`; therefore, a formula of the form `firable`^{J}`(stack(X,Y))` should be represented as `firable`^{J}`(stack,X,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 $\mathcal{CH}(\Pi)$. Datalog$_{1S}$ programs are known to have interesting and decidable formal properties [5], which follow from the fact that their temporal arguments are isomorphic to Presburger arithmetic and their nontemporal arguments range over a finite *reduced Herbrand base* defined as follows. Given a Datalog$_{1S}$ program P consider the program P' obtained by removing the $1S$ argument from P. P' is a Datalog program without any function symbols, thus it has a finite Herbrand Base which we will call *reduced Herbrand base* for P. This notion is used in Lemma 10, below, to derive a complexity bound for the computation of a stable model for $\mathcal{CH}(\Pi)$. Observe that XY-stratification holds both for $\mathcal{CH}(\Pi)$ and its Datalog$_{1S}$ counterpart with reduced Herbrand base $B'_{\mathcal{CH}(\Pi)}$. Then, directly from the definitions we have that:

Lemma 10 *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 choice program,
(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 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 $N = |B'_{\mathcal{CH}(\Pi)}|$. Because of the many performance improvements due to XY-stratification, the coefficient and degree of the polynomial can in fact be kept very small, and the construction of each single candidate plan is done quite efficiently by systems such as $\mathcal{LDL}++$ and Aditi (in particular $\mathcal{LDL}++$ will be able to recognize that the frame axiom rules are best implemented by simply updating the old state rather than copying them into a new state [35]).

Nevertheless, the number of alternative plans (i.e., choice models) normally grows exponentially with the length of the plans. The notions of partially ordered plans and systematicity discussed in the next section can be used to reduce such a growth. Another solution approach consists in adding some heuristic function to guide the search. Thus, a heuristic measure, H (e.g., an estimate of the distance to the goal) is associated at each choice step with each firable action α. Then, the choice rule (3) becomes:

$$\texttt{fired}^{\texttt{J}}(\alpha) \leftarrow \ \texttt{firable}^{\texttt{J}}(\alpha, \texttt{H}), \neg\texttt{firable}^{\texttt{J}}(\texttt{end}, \_),$$
$$\texttt{choice}^{\texttt{J}}(\alpha), \texttt{choice\_least}^{\texttt{J}}(\texttt{H}).$$

Where, $\texttt{choice\_least}^{\texttt{J}}(\texttt{H})$ denotes that, rather than selecting an arbitrary H for each J, as in the standard choice construct, we must select one that has the least value for H. This simple specialization of choice leads to the expression and implementation in Datalog of greedy algorithms, such as Dijkstra's or Prim's algorithms [13]. For planning problems it produces support for greedy heuristics within the XY-stratification framework.

Another planning problem that can be implemented easily in our framework is that of checking whether a given sequence of actions is in fact a valid plan. Indeed, let a set of facts,

$$\texttt{cplan}(\texttt{J}, \alpha)$$

represent a sequence of actions that describe our candidate plan. Obviously, there must be a start pair, $(0, \texttt{begin})$, and a final pair $(\texttt{n}, \texttt{end})$, where $\texttt{n}$ is the largest sequence number in the first column. Then, we can simply add the goal $\texttt{cplan}^{\texttt{J}}(\alpha)$ to rule (3) and $\texttt{cplan}^{\texttt{J}}(\texttt{end})$ to rule (4). Then our candidate plan is actually a valid plan iff the stable model resulting from the computation of our XY-program contains $\texttt{fired}^{\texttt{n}}(\texttt{end})$. Furthermore, if the stable model contains $\texttt{fired}^{\texttt{j}}$ but not $\texttt{fired}^{\texttt{j}+1}$ we can conclude that the first j steps of this candidate plan are valid, but the $j+1$ step is not.

Typically, checking the validity of a given plan was the main operation supported by the logic-based formalisms proposed in the past. Our approach also supports the generation of plans.

5 Parallel Plans

In this section we show how the model presented in the previous sections can be naturally extended to characterize *parallel* plans. A large variety of problems require generating parallel execution

plans. In multi-agent systems, for instance, parallel activities are part of the planning domain itself.

We first introduce a simple notion of parallel plan. We then discuss an effective approach to the problem of generating parallel execution plans by converting given totally ordered plans into equivalent parallel plans. We show how a simple compression technique for transforming totally ordered plans into equivalent parallel plans can be incorporated into our model. We then show that the proposed parallelization technique can be applied also to partially ordered plans, which represent sets of totally ordered plans. Finally, we show that the proposed parallelization technique can be fruitfully employed for defining a systematic search strategy for (parallel) plans.

5.1 Parallel Plans: Definition

A parallel plan is a sequence $< \Gamma_1; \ldots; \Gamma_m >$ where each Γ_i is a set of actions to be executed in parallel (rather than a single action as in the case of totally ordered, sequential plans). In order to formally define the notion of parallel plans, it is necessary to establish *what* is the overall effect of the parallel execution of a set of actions, and *when* a set of actions can be executed in parallel.

Several notions of parallel plans have been proposed in the literature. Some of these notions take into account the possibly different duration of actions or cooperating simultaneous actions, whose effect cannot be always obtained by their sequential execution [1, 12, 16]. We consider here the simpler situation in which the effects of the parallel execution of a set of actions is defined as a combination of the effects of each action in the set. Simply stated, the behavior of a set of actions $\{\alpha_1, \ldots, \alpha_n\}$ may be described as a single parallel action defined as follows:

$$
\begin{aligned}
Pre(\{\alpha_1, \ldots, \alpha_n\}) &= Pre(\alpha_1) \cup \ldots \cup Pre(\alpha_n) \\
Post(\{\alpha_1, \ldots, \alpha_n\}) &= Post(\alpha_1) \cup \ldots \cup Post(\alpha_n)
\end{aligned}
$$

Actions may however interfere each other, and it is therefore necessary to avoid or rule such possible interferences. The *independence* criterion has been largely employed to restrict the class of actions executable in parallel (e.g., [18, 24, 26]). Simply stated, a set of actions can be executed in parallel only if their parallel execution produces the same result as *every* serialization of these actions. Notice that the above criterion of serializability defines a strong notion of independent actions. For instance consider the problem of relocating two different blocks in different areas of a blocks world with only one robot. While the order in which the actions are performed may be irrelevant, the actions cannot be executed in parallel since not all their serializations are executable plans.

Definition 5 *A set of ground actions $\{\alpha_1, \ldots, \alpha_n\}$ is executable in a state S, resulting in a state S' if and only if:*

(1) $S \models Pre(\alpha_1) \cup \ldots \cup Pre(\alpha_n)$ *and*

(2) *For each sequence $< \beta_1; \ldots; \beta_n >$ that is a permutation of $\{\alpha_1, \ldots, \alpha_n\}$ there exists a sequence of states $T_1, \ldots, T_{n-1}$:*

$$
S \overset{\beta_1}{\Longrightarrow} T_1 \ldots \overset{\beta_{n-1}}{\Longrightarrow} T_{n-1} \overset{\beta_n}{\Longrightarrow} S'
$$

where $S' = (S - (Del(\beta_1) \cup \ldots \cup Del(\beta_n))) \cup (Add(\beta_1) \cup \ldots \cup Add(\beta_n))$.

Note that in the above definition substitutions are omitted (e.g., in $S \overset{\beta_1}{\Longrightarrow} T_1$) as we are dealing with ground actions only. Let us now introduce some syntactic conditions on (sets of) actions that ensure that a set of actions is executable (in parallel).

16

Definition 6 *Let α, β be two ground actions. We say that the pair of actions (α, β) is* compatible *if and only if the following sets of literals are non-contradictory:*

(1) $Pre(\alpha)$ and $Post(\beta)$,
(2) $Pre(\beta)$ and $Post(\alpha)$,
(3) $Post(\alpha)$ and $Post(\beta)$.

A set of ground actions $\{\alpha_1, \ldots, \alpha_n\}$ is well-formed *if and only if each pair (α_i, α_j), where $i \neq j$, is compatible.*

The well-formedness condition establishes when a set of actions is executable in parallel in a state S which satisfies the preconditions of these actions.

Proposition 11 *Let $\alpha_1, \ldots, \alpha_n$ be ground actions, and let S be a state such that $S \models Pre(\alpha_1) \cup \ldots \cup Pre(\alpha_n)$. Then:*

$$\{\alpha_1, \ldots, \alpha_n\} \text{ is well-formed} \iff \{\alpha_1, \ldots, \alpha_n\} \text{ is executable in } S.$$

Proof. ($\Longrightarrow$) Let $< \beta_1; \ldots; \beta_n >$ be a permutation of $\{\alpha_1, \ldots, \alpha_n\}$. Since $S \models Pre(\beta_1)$ then $\exists T_1 : S \overset{\beta_1}{\Longrightarrow} T_1$. Moreover, since $S \models Pre(\beta_2)$ and since $Post(\beta_1)$ and $Pre(\beta_2)$ are non-contradictory (since β_1 and β_2 are compatible) then $T_1 \models Pre(\beta_2)$. Therefore $\exists T_1, T_2 : S \overset{\beta_1}{\Longrightarrow} T_1 \overset{\beta_2}{\Longrightarrow} T_2$. We now observe that since β_1 and β_2 are compatible then $Add(\beta_1) \cap Del(\beta_2) = \emptyset$ and hence $((S - Del(\beta_1)) \cup Add(\beta_1)) - Del(\beta_2) = ((S - Del(\beta_1)) - Del(\beta_2)) \cup Add(\beta_1)$. Therefore $T_2 = (((S - Del(\beta_1)) \cup Add(\beta_1)) - Del(\beta_2)) \cup Add(\beta_1)$ is equal to $(S - (Del(\beta_1) \cup Del(\beta_2))) \cup (Add(\beta_1) \cup Add(\beta_2))$. By applying the same argument to $\beta_3, \beta_4, \ldots, \beta_n$ we have that $\exists T_1, \ldots, T_{n-1} : S \overset{\beta_1}{\Longrightarrow} T_1 \ldots \overset{\beta_{n-1}}{\Longrightarrow} T_{n-1} \overset{\beta_n}{\Longrightarrow} S'$ where $S' = (S - (Del(\beta_1) \cup \ldots \cup Del(\beta_n))) \cup (Add(\beta_1) \cup \ldots \cup Add(\beta_n))$.

($\Longleftarrow$) We now show, by counter-positive, that if $\{\alpha_1, \ldots, \alpha_n\}$ is not well-formed then $\{\alpha_1, \ldots, \alpha_n\}$ is not executable in S. Suppose that α_i and α_j are not compatible. If $Pre(\alpha_i)$ and if $Post(\alpha_j)$ are contradictory then for all sequences $< \beta_1; \ldots; \beta_n >$ s.t. $\beta_{n-1} = \alpha_j$ and $\beta_n = \alpha_i$ there is no S' s.t. $S \overset{\beta_1; \ldots; \beta_n}{\Longrightarrow} S'$ since β_n is not executable after β_{n-1} and hence $\{\alpha_1, \ldots, \alpha_n\}$ is not executable in S. If instead there exists x s.t. $x \in Post(\alpha_i)$ and $\neg x \in Post(\alpha_j)$ then for all sequences $< \beta_1; \ldots; \beta_n >$ s.t. there exists S' s.t. $S \overset{\beta_1; \ldots; \beta_n}{\Longrightarrow} S'$ we have that if $\beta_{n-1} = \alpha_i$ and $\beta_n = \alpha_j$ then $x \notin S'$, while if $\beta_{n-1} = \alpha_j$ and $\beta_n = \alpha_i$ then $x \in S'$. Hence $\{\alpha_1, \ldots, \alpha_n\}$ is not executable in S. $\square$

In the next sections, we shall present a technique for converting a totally ordered, sequential plan into an equivalent parallel plan. The technique relies on an analysis of the *causal links* of a plan. The notion of causal link was introduced by Tate [29] in the NONLIN planner as a means for representing the causal dependencies among 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 $\alpha \overset{x}{\to} \beta$.

Definition 7 *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:*
$$\mathcal{CL}(\pi) = \{\alpha_i \overset{x}{\to} \alpha_{i+j} \mid x \in Post(\alpha_i) \wedge x \in Pre(\alpha_{i+j}) \wedge \nexists \alpha_h : (x \in Post(\alpha_h) \wedge i < h < i+j)\}.$$

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

Actions can also *inhibit* each other. We say that an action *inhibits* another action if the former falsifies some of the preconditions of the latter, as formalized by the following definition.

Definition 8 *Let α and β be two ground actions. We say that α* inhibits *β if and only if*

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

Thus, α inhibits β either if 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 8 and in the sequel we use the notational convention that $\neg(\neg x) = x$.

We will next introduce the notion of *strict* action, which simplifies the treatment of parallel plans without losing generality. The *non-strict* interpretation 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 α must also belong to the positive preconditions of α, so as to ensure that no deletion will be interpreted as a null action. The symmetric observation holds for additions. Roughly speaking, an action that deletes (resp., adds) x may be executed in a state S only if x belongs (resp., does not belong) to S.

Definition 9 *An action α is* strict *if and only if $\forall x: \ (x \in Post(\alpha) \Longrightarrow \neg x \in Pre(\alpha))$.*

Namely an action is strict if $\forall x: \ ((x \in Del(\alpha) \implies x \in Pre(\alpha)) \ \wedge \ (x \in Add(\alpha) \implies \neg x \in Pre(\alpha)))$. Note that the syntactic condition of strictness does not lead to any loss of generality, because the non-strict interpretation of actions can be expressed by via strict actions. For instance, suppose α is not strict because of the existence of 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.

The strictness condition simplifies the treatment of parallel plans: Whenever the preconditions of two strict actions α and β hold in a state S, then the sets of postconditions of the two actions are compatible. This implies that the third condition in Definition 6 need not be tested when checking the compatibility of two actions whose preconditions hold in the current state. Therefore, we will use a strict interpretation of actions and refer to strict plans and planning problems in the obvious way.

Example 3 (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), ¬hand_empty(R), holding(R,X)}`

```
putdown(R,X): Robot R puts block X on the table.
Pre =   {holding(R,X), ¬ontable(X), ¬hand_empty(R) }
Post =  {¬holding(R,X), ontable(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), ¬hand_empty(R) }
Post =  {¬clear(Y), ¬holding(R,X), 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 =  {¬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}
```

For instance, the totally ordered plan:

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

is a solution for Π.

5.2 Parallelization of Totally Ordered Plans

We now present a first simple compression technique for parallelizing a given totally ordered plan. Since each totally ordered plan $< \alpha_1; \ldots; \alpha_n >$ can be viewed as a parallel plan $< \{\alpha_1\}; \ldots; \{\alpha_n\} >$, the technique can be defined in terms of compression steps on parallel plans.

The basic underlying idea is the following. If α and β are two consecutive actions in a plan π and if (1) α does not cause β in π, and (2) β does not inhibit α, then α and β can be executed in parallel without affecting the result of the overall plan. The above intuition is formalized by the following notion of *compression step* on a plan.

Definition 10 *Let* $\pi = < \Gamma_1; \ldots; \Gamma_{i-1}; \{\alpha_1, \ldots, \alpha_n\}; \{\beta\}; \Gamma_{i+2}; \ldots; \Gamma_m >$ *be a plan. We say that* $\pi' = < \Gamma_1; \ldots; \Gamma_{i-1}; \{\alpha_1, \ldots, \alpha_n, \beta\}; \Gamma_{i+2}; \ldots; \Gamma_m >$ *is obtained from* π *via a* compression step *if and only if* $\forall i \in [1, n]$: *(1)* α_i *does not cause* β, *and (2)* β *does not inhibit* α_i.

It is worth noting that compression steps preserve plan equivalence. Let us consider the following (strong) notion of plan equivalence: Two plans are equivalent w.r.t. a state S if and only if they lead from S to the same final state.

Lemma 12 *Let π be a plan for $\Pi =< S, G, \mathcal{A} >$, and let π' be obtained from π via a compression step. Then π' is a plan for Π, and π and π' are equivalent w.r.t. S.*

Proof. Let $\pi =< \Gamma_1; \ldots; \Gamma_{i-1}; \{\alpha_1, \ldots, \alpha_n\}; \{\beta\}; \Gamma_{i+2}; \ldots; \Gamma_m >$ and let $\pi' = < \Gamma_1; \ldots; \Gamma_{i-1}; \{\alpha_1, \ldots, \alpha_n, \beta\}; \Gamma_{i+2}; \ldots; \Gamma_m >$ be obtained from π via a compression step. We first show that the set of actions $\{\alpha_1, \ldots, \alpha_n, \beta\}$ is well-formed. The set $\{\alpha_1, \ldots, \alpha_n\}$ is well-formed, since π is a plan for Π. Suppose now that $\{\alpha_1, \ldots, \alpha_n, \beta\}$ is not well-formed. This means that $\exists \alpha_i : \alpha_i$ and β are not compatible. If $Pre(\beta)$ and $Post(\alpha_i)$ are contradictory then β is not executable after α_i, but this contradicts the hypothesis that π is a plan for Π. If $Pre(\alpha_i)$ and $Post(\beta)$ are contradictory then β inhibits α_i, while if $Post(\alpha_i)$ and $Post(\beta)$ are contradictory then, by strictness, α_i causes β. In both cases π' could not be obtained from π via a compression step. Contradiction. Therefore the set of actions $\{\alpha_1, \ldots, \alpha_n, \beta\}$ is well-formed. Finally, by Proposition 11, we have that the set $\{\alpha_1, \ldots, \alpha_n, \beta\}$ is executable in the state produced by the sequence of actions $\pi' =< \Gamma_1; \ldots; \Gamma_{i-1} >$ starting from S. Hence π' is a plan for Π, and π and π' are equivalent w.r.t. S. $\square$

Compression steps may be applied repeatedly. We say that a plan π' is a *compression* of a plan π iff π' is obtained from π by means of a (finite) sequence of compression steps. It is easy to observe that the definition of compression step suggests that the best way to compress a plan is to apply all possible compression steps while visiting the plan in a left-to-right order.

We now present the Datalog_{1S} rules for modelling the parallelization of a totally ordered plan. The causality relation between two actions α and β can be defined as follows:

$$\texttt{prec}(\alpha, \beta) \quad \leftarrow \quad \texttt{postcond}(\alpha, \texttt{X}, \texttt{Type}), \texttt{precond}(\beta, \texttt{X}, \texttt{Type}). \tag{13}$$

where the relation `precond` is defined analogously to `postcond` (Section 3). The above rule states that action α establishes some condition X needed by β for firing. Before a plan is actually constructed, this rule only establishes patterns 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:

$$\texttt{opposite}(\texttt{pos}, \texttt{neg}). \tag{14}$$
$$\texttt{opposite}(\texttt{neg}, \texttt{pos}). \tag{15}$$

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

$$\texttt{prec}(\alpha, \beta) \quad \leftarrow \quad \texttt{precond}(\alpha, \texttt{X}, \texttt{Type}), \texttt{postcond}(\beta, \texttt{X}, \texttt{UnType}), \tag{16}$$
$$\texttt{opposite}(\texttt{Type}, \texttt{Untype}).$$

The process of parallelizing (by compressing) a totally ordered plan π actually consists of partitioning the actions of π into sets of actions or layers to be executed in parallel. Consider for instance the totally ordered plan π_1 introduced in Example 4 and represented by the second column of the following table:

Stage	π_1	Groups
0	$\mathtt{fired}^0(\mathtt{begin})$	$\mathtt{new}^0$
1	$\mathtt{fired}^1(\mathtt{unstack(r1,c,a)})$	$\mathtt{new}^1$
2	$\mathtt{fired}^2(\mathtt{putdown(r1,c)})$	$\mathtt{new}^2$
3	$\mathtt{fired}^3(\mathtt{pickup(r1,a)})$	$\mathtt{new}^3$
4	$\mathtt{fired}^4(\mathtt{pickup(r2,b)})$	
5	$\mathtt{fired}^5(\mathtt{stack(r2,b,c)})$	$\mathtt{new}^5$
6	$\mathtt{fired}^6(\mathtt{stack(r1,a,b)})$	$\mathtt{new}^6$
7	$\mathtt{fired}^7(\mathtt{end})$	$\mathtt{new}^7$

The third column of the table represents a partition of the actions of π_1 into groups of parallel actions. Namely, each action corresponds to a singleton set, except for the two `pickup` actions introduced at stages 3 and 4 which are grouped together. Predicate `new` is employed to represent the grouping of actions corresponding to a left-to-right compression of plan π_1:

```
<begin;unstack(r1,c,a);putdown(r1,c); {pickup(r1,a),pickup(r2,b)};
 stack(r2,b,c);stack(r1,a,b);end>
```

In order to give the Datalog$_{1S}$ rules for parallelizing a given totally ordered plan, we therefore define predicate `new` that marks the beginning of a new set of parallel actions. To this end, we will also use the predicate $\mathtt{curr\_strat}^\mathsf{J}(\alpha)$, which accumulates the actions in the current layer.

Simply stated, the firing of an action β starts a new layer if there is some action α in the current layer that precedes β. Moreover, the first layer always starts at stage 0.

$$\mathtt{new}^0. \tag{17}$$

$$\mathtt{new}^\mathsf{J} \quad \leftarrow \quad \mathtt{fired}^\mathsf{J}(\beta), \mathtt{curr\_strat}^\mathsf{J}(\alpha), \mathtt{prec}(\alpha, \beta). \tag{18}$$

The predicate `curr_strat` is defined by the following rules:

$$\mathtt{curr\_strat}^{\mathsf{J}+1}(\alpha) \quad \leftarrow \quad \mathtt{curr\_strat}^\mathsf{J}(\alpha), \neg\mathtt{new}^\mathsf{J}. \tag{19}$$

$$\mathtt{curr\_strat}^{\mathsf{J}+1}(\alpha) \quad \leftarrow \quad \mathtt{fired}^\mathsf{J}(\alpha). \tag{20}$$

5.3 Parallelization of Partially Ordered Plans

The parallelization method presented in the previous section can be used to construct a *minimal* (i.e., with a minimal number of parallel actions) compression of a totally ordered plan. It is easy to see, however, that there may be shorter (i.e., with a smaller number of parallel actions) parallel plans among all variants of a plan. For instance, given a totally ordered plan π, there may be a shorter parallel plan which is not a compression of π but which is a compression of some variant of π, in which the actions of π are executed in a different order.

The parallelization technique presented in the previous section can be extended so as to find, given a totally ordered plan π, a minimal parallel variant of π which has the same causal links as π. The basic idea is to consider a set of plans (namely the partially ordered abstraction of the given totally ordered plan), represented by a set of precedence constraints among actions. Then the process of parallelizing the obtained (partially ordered) plan can be simply described by applying standard graph algorithms.

We now introduce a relation $\prec$ that defines a partial order on actions and models the partially ordered abstraction of a plan. Intuitively speaking, $\alpha \prec \beta$ holds if the action α must precede β in the plan.

Definition 11 *Let $\pi =< \alpha_1;\dots;\alpha_n >$ be a totally ordered plan, and let α_i, α_{i+h} $(h > 0)$ be two actions in π. The set of constraints $Ab(\pi)$ is defined as follows:*

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

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

Definition 12 *Let $\pi =< \alpha_1;\dots;\alpha_n >$ be a totally ordered plan. A* linearization *of $Ab(\pi)$ is a permutation $< \beta_1;\dots;\beta_n >$ of $< \alpha_1;\dots;\alpha_n >$ that satisfies $Ab(\pi)$— i.e., such that $\nexists i, h:\ h > 0 \wedge \beta_{i+h} \prec \beta_i \in Ab(\pi)$.*

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 a directed acyclic graph. Let $\mathcal{A}_\pi$ be the set of actions occurring in a totally ordered plan π. Then the pair $(\mathcal{A}_\pi, Ab(\pi))$ can be represented by a directed acyclic graph where the nodes are the actions $\mathcal{A}_\pi$ occurring 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 each linearization of $Ab(\pi)$ simply corresponds to a topological sort for the corresponding graph $(\mathcal{A}_\pi, Ab(\pi))$. Moreover, the set of constraints $Ab(\pi)$ represents a set of equivalent plans, as established by the following proposition.

Proposition 13 *Let π be a totally ordered plan for $\Pi =< S, \mathcal{A}, G >$. Then for each linearization π' of $Ab(\pi)$: π' is a totally order plan for Π, and π' and π are equivalent w.r.t. S.*

Proof. Let $\pi =< \alpha_1;\dots;\alpha_n >$ and suppose that $\pi' =< \beta_1;\dots;\beta_n >$ is a linearization of $Ab(\pi)$ which is not a totally ordered plan for Π. Then $\exists S_1, S_2,\dots, Si-1$ such that: $S \overset{\beta_1}{\Longrightarrow} S_1 \overset{\beta_2}{\Longrightarrow} S_2 \dots \overset{\beta_{i-1}}{\Longrightarrow} S_{i-1}$, and $S_{i-1} \not\models Pre(\beta_i)$, i.e., $\exists x:\ x \in Pre(\beta_i) \wedge S_{i-1} \not\models x$. Since π' is a linearization of $Ab(\pi)$ there exists $h \in [1, n]$: $\alpha_h = \beta_i$, and by strictness, there exists k: $(\alpha_{h-k} \overset{L}{\to} \alpha_h) \in \mathcal{CL}(\pi)$. Moreover, by definition of $Ab(\pi)$, there exists j: $\alpha_{h-k} = \beta_{i-j}$. Then, since $S_{i-1} \not\models x$, there exists z such that $z \in (i - j, i)$ and $\neg x \in Post(\beta_z)$. 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 $\alpha_{h-k} \overset{L}{\to} \alpha_h \in \mathcal{CL}(\pi)$. Moreover $m \nless h - k$ (otherwise $\beta_z \prec \beta_{i-j} \in Ab(\pi)$), and $m \ngtr h$ (otherwise $\beta_i \prec \beta_z \in Ab(\pi)$). Hence there is no $m \in [1, n]$: $\beta_z = \alpha_m$. Contradiction. Finally if π' is a permutation of π and both π' and π are totally ordered plans for Π, then — by strictness — they produce the same final state. $\square$

Corollary 14 *Let π be a totally ordered solution for Π. Then each linearization of $Ab(\pi)$ is a totally ordered solution for Π.*

Most importantly, the set of constraints $Ab(\pi)$ also represents a set of *parallel* plans. These can be characterized by means of a general notion of multi-level sort.

Definition 13 *Let $\mathcal{O}$ be a set of constraints of the form $\alpha \prec \beta$, where α and β belong to a set $\mathcal{A}$ of elements. A partition $< \Gamma_1;\dots;\Gamma_m >$ of $\mathcal{A}$ is a* multi-level sort *of $(\mathcal{A}, \mathcal{O})$ if and only if $(\alpha \prec \beta \in \mathcal{O} \wedge \beta \in \Gamma_i \wedge \alpha \in \Gamma_j) \implies i < j$.*

Given a totally ordered plan π, each multi-level sort of $(\mathcal{A}, \mathcal{O})$ is a compression of some linearization of $Ab(\pi)$, and vice-versa.

Proposition 15 *Let π be a totally ordered plan, and let $\mathcal{A}_\pi$ be the set of actions occurring in π. Then:*

$$\pi' \text{ is a multi-level sort of } (\mathcal{A}_\pi, Ab(\pi)) \iff \begin{array}{l} (1)\ \pi' \text{ is a compression of } \pi'', \text{ and} \\ (2)\ \pi'' \text{ is a linearization of } Ab(\pi). \end{array}$$

Proof. ($\Longrightarrow$) Let $\pi' = < \Gamma_1; \ldots; \Gamma_m >$ be a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. Consider the set of actions $\Gamma_1 = \{\alpha_1, \ldots, \alpha_{h_1}\}$ in π'. By hypothesis, we have that $\forall i, j \in [1, h_1]: \alpha_i \prec \alpha_j \notin Ab(\pi)$, that is for each pair of actions α_i, α_j in Γ_1 α_i neither causes nor inhibits α_j. Therefore the set Γ_1 can be obtained by means of $(h_1 - 1)$ compression steps from any sequence $< \beta_1; \ldots; \beta_{h_1} >$ which is a permutation of $\{\alpha_1, \ldots, \alpha_{h_1}\}$. The same observation applies to the other sets of actions $\Gamma_2, \ldots, \Gamma_m$ in π'. Therefore π' is a compression of π'' for any $\pi'' = < \gamma_1^1; \ldots; \gamma_{h_1}^1; \gamma_1^2; \ldots; \gamma_{h_2}^2; \ldots \gamma_1^m; \ldots; \gamma_{h_m}^m >$ such that $\{\gamma_1^i; \ldots; \gamma_{h_i}^i\} = \Gamma_i$, for $i \in [1, m]$. It is easy to observe that any such π'' is a linearization of $Ab(\pi)$. Indeed this is not the case if either $(\exists \gamma^a, i, \gamma_j^b \in \pi'' : a < b \wedge \gamma_j^b \prec \gamma_i^a \in Ab(\pi))$ or $(\exists \gamma^a, i, \gamma_j^a \in \pi'' : i < j \wedge \gamma_j^b \prec \gamma_i^a \in Ab(\pi))$. In either case, π' would not be a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. Contradiction.

($\Longleftarrow$) Suppose that $\pi' = < \Gamma_1; \ldots; \Gamma_m >$ is a compression of π'', with π'' linearization of $Ab(\pi)$, and that π' is not a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. This means that $\exists \alpha, \beta, i, j: \alpha \prec \beta \in Ab(\pi) \wedge \alpha \in \Gamma_i \wedge \beta \in \Gamma_j \wedge i \geq j$. Now if $i = j$ then π' is not a compression, while if $i > j$ then π'' is not a linearization of $Ab(\pi)$. In both cases we hence obtain a contradiction. $\square$

Proposition 15 therefore establishes that the problem of finding a minimal parallel plan among all possible compressions of the linearizations of $Ab(\pi)$ reduces to finding a minimal multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. The problem of determining a minimal multi-level sort of a pair $(\mathcal{A}_\pi, Ab(\pi))$ can be solved by applying standard graph algorithms. Let G be the directed acyclic graph representing $(\mathcal{A}_\pi, Ab(\pi))$. Then $< \Gamma_1; \ldots; \Gamma_m >$ is a minimal multi-level sort for $(\mathcal{A}_\pi, Ab(\pi))$ if and only if each set Γ_{i+1} is the set of all nodes in G with maximal distance i from the source (viz., node `begin`).

It is easy to see that the above described parallelization technique can be modeled by means of Datalog$_{1S}$ rules. Rather than illustrating this, we will discuss how the technique can be exploited to improve the search strategy for (parallel) plans. This is the scope of the next section.

5.4 Systematic Search

We first observe that the set of constraints $Ab(\pi)$ defines the partially ordered abstraction of a plan π, that is the mapping from π to $Ab(\pi)$ defines the inverse operations of finding a linearization of a partially ordered plan.

Proposition 16 *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. $\qquad\square$

Proposition 13 and 16 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 which is 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.

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 14 *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) $Layer(\alpha_j) \le Layer(\alpha_{j+k})$, 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.

Example 4 *Consider again the planning problem described in Example 4. 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:*

$\pi_1 =$ `<unstack(r1,c,a);putdown(r1,c);pickup(r1,a);`
        `pickup(r2,b);stack(r2,b,c);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 (superscripts are omitted since there is no repetition of the same action in the original plan):*

$\text{unstack}(r1,c,a)$	$\prec$	$\text{putdown}(r1,c)$	$\text{putdown}(r1,c)$	$\prec$	$\text{stack}(r1,a,b)$
$\text{unstack}(r1,c,a)$	$\prec$	$\text{pickup}(r1,a)$	$\text{pickup}(r1,a)$	$\prec$	$\text{stack}(r1,a,b)$
$\text{unstack}(r1,c,a)$	$\prec$	$\text{stack}(r2,b,c)$	$\text{pickup}(r2,b)$	$\prec$	$\text{stack}(r2,b,c)$
$\text{unstack}(r1,c,a)$	$\prec$	$\text{stack}(r1,a,b)$	$\text{pickup}(r2,b)$	$\prec$	$\text{stack}(r1,a,b)$
$\text{putdown}(r1,c)$	$\prec$	$\text{pickup}(r1,a)$	$\text{stack}(r2,b,c)$	$\prec$	$\text{stack}(r1,a,b)$
$\text{putdown}(r1,c)$	$\prec$	$\text{stack}(r2,b,c)$			

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

$\pi_2 =$ `<pickup(r2,b);unstack(r1,c,a);putdown(r1,c);`
        `stack(r2,b,c);pickup(r1,a);stack(r1,a,b)>`

$\pi_3 =$ `<unstack(r1,c,a);pickup(r2,b);putdown(r1,c);`
        `stack(r2,b,c);pickup(r1,a);stack(r1,a,b)>`

$\pi_4 =$ `<unstack(r1,c,a);pickup(r2,b);putdown(r1,c);`
        `pickup(r1,a);stack(r2,b,c);stack(r1,a,b)>`

$\pi_5 =$ `<unstack(r1,c,a);putdown(r1,c);pickup(r2,b);`
        `stack(r2,b,c);pickup(r1,a);stack(r1,a,b)>`

$\pi_6 =$ `<pickup(r2,b);unstack(r1,c,a);putdown(r1,c);`
        `pickup(r1,a);stack(r2,b,c);stack(r1,a,b)>`

$\pi_7 =$ `<unstack(r1,c,a);putdown(r1,c);pickup(r2,b);`
        `pickup(r1,a);stack(r2,b,c);stack(r1,a,b)>`

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

We now present 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 $\pi = <$ `begin`$; \alpha_1; \ldots; \alpha_{n-1} >$ where $Layer(\alpha_{n-1}) = L$. Let 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 $\pi = <\texttt{begin}; \alpha_1; \ldots; \alpha_{n-1}>$ 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 $\pi' = <\texttt{begin}; \alpha_1; \ldots; \alpha_{n-1}; \beta>$ 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 now present the Datalog$_{1S}$ rules for constructing a canonical plan. We extend the description given in Sections 3 and 5.2. We use the relations `prec` and `opposite` as defined by rules (13)—(16). Moreover, in addition to predicate `curr_strat` (defined by rules (19) and (20)), we will use a predicate $\texttt{last\_strat}^{J}(\alpha)$ which 'remembers' the actions in the previous layer.

The rules defining the `firable` actions remain the same as in the case of totally ordered plans (Section 3). However, the old rule for `fired` (3) specifying that a choice can be made at random from the actions 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.

A firable action can be added to the plan as part of the next layer as follows:

$$\texttt{next}^{J}(\beta) \quad \leftarrow \quad \texttt{firable}^{J}(\beta), \texttt{curr\_strat}^{J}(\alpha), \qquad (21)$$
$$\texttt{prec}(\alpha, \beta).$$

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

$$\texttt{same}^{J+1}(\beta) \quad \leftarrow \quad \texttt{firable}^{J+1}(\beta), \neg\texttt{next}^{J+1}(\beta), \qquad (22)$$
$$\texttt{last\_strat}^{J+1}(\alpha), \texttt{prec}(\alpha, \beta),$$
$$\texttt{fired}^{J}(\gamma), \texttt{follows}(\beta, \gamma).$$

In other words, a firable 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 ($\texttt{follows}(\beta, \gamma)$), according to a lexicographical order.

Finally, the predicate `last_strat` is defined by the following rules:

$$\texttt{last\_strat}^{J+1}(\alpha) \quad \leftarrow \quad \texttt{last\_strat}^{J}(\alpha), \neg\texttt{new}^{J}. \qquad (23)$$
$$\texttt{last\_strat}^{J+1}(\alpha) \quad \leftarrow \quad \texttt{curr\_strat}^{J}(\alpha), \texttt{new}^{J}. \qquad (24)$$

It is worth observing that, in terms of XY-stratification, the heads of the rules are always interpreted as 'new' values. Thus $\texttt{fired}^{J}(\gamma)$ in the rule (22) refers to an old value, while $\texttt{next}^{J}(\texttt{B})$ and $\texttt{same}^{J+1}(\texttt{B})$ in the rules (21) and (22) refer to new values. Rules (21) and (22) can be now combined to define the new subset of firable actions as follows:

$$\texttt{subfirable}^{J}(\beta) \quad \leftarrow \quad \texttt{same}^{J}(\beta). \qquad (25)$$
$$\texttt{subfirable}^{J}(\beta) \quad \leftarrow \quad \texttt{next}^{J}(\beta). \qquad (26)$$

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

$$\texttt{fired}^{\mathsf{J}}(\alpha) \quad \leftarrow \quad \texttt{subfirable}^{\mathsf{J}}(\alpha), \texttt{choice}^{\mathsf{J}}(\alpha). \tag{27}$$

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

$$\texttt{fired}^{0}(\alpha) \quad \leftarrow \quad \texttt{firable}^{0}(\alpha), \texttt{choice}^{0}(\alpha). \tag{28}$$

When applied to the problem at hand, the new set of systematic rules yields the following plan (which corresponds to the plan produced by a minimal compression of π_6):

Stage	π_6	Groups
0	$\texttt{fired}^{0}(\texttt{begin})$	$\texttt{new}^{0}$
1	$\texttt{fired}^{1}(\texttt{pickup(r2, b)})$	$\texttt{new}^{1}$
2	$\texttt{fired}^{2}(\texttt{unstack(r1, c, a)})$	
3	$\texttt{fired}^{3}(\texttt{putdown(r1, c)})$	$\texttt{new}^{3}$
4	$\texttt{fired}^{4}(\texttt{pickup(r1, a)})$	$\texttt{new}^{4}$
5	$\texttt{fired}^{5}(\texttt{stack(r2, b, c)})$	
6	$\texttt{fired}^{6}(\texttt{stack(r1, a, b)})$	$\texttt{new}^{6}$
7	$\texttt{fired}^{7}(\texttt{end})$	$\texttt{new}^{7}$

Also observe that the program for systematic parallelization, albeit more complex, remains XY-stratified as shown in Figure 3.
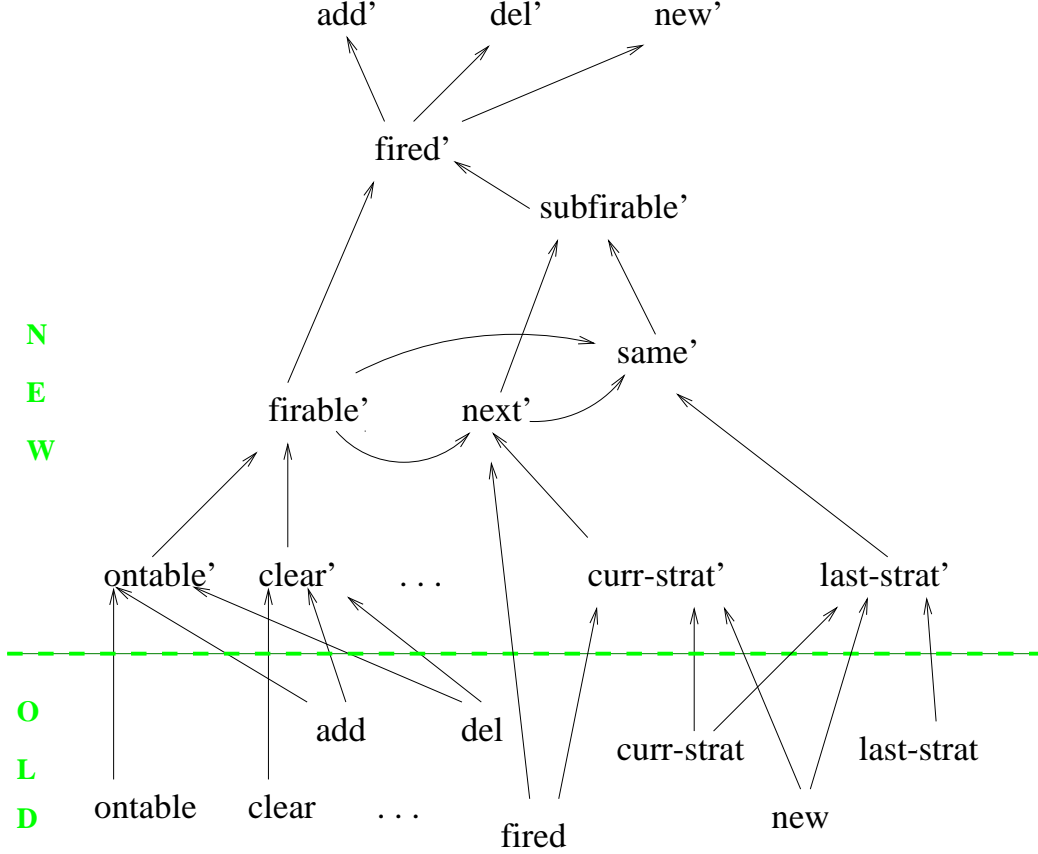
The search strategy that we have described is systematic in the space of totally ordered plans. Indeed, as we have observed, the universe of totally ordered plans can be partitioned into equivalence classes, where two plans are equivalent if and only if they have the same partially ordered abstraction. Since each partially ordered plan $Ab(\pi)$ has exactly one canonical linearization, the search in the space of canonical plans is systematic in the sense that at most one totally ordered plan for each equivalence class will be found.

It is important to observe that such a search strategy is also systematic in the space of parallel plans. Indeed, the partially ordered abstraction $Ab(\pi)$ of a plan π represents a set of parallel variants of π (Proposition 15). This means that the notion of partially ordered abstraction partitions the universe of totally ordered *and* parallel plans into equivalence classes, where each class contains all (possibly parallel) causal-link preserving variants of each plan in the class.

Notably, a canonical plan also represents a *minimal* (i.e., with a minimal number of parallel actions) parallel plan in the equivalence class. Actually, the *canonical parallelization* of $(\mathcal{A}_\pi, Ab(\pi))$ can be determined from the canonical linearization π' of $(\mathcal{A}_\pi, Ab(\pi))$ by grouping together all the sequences of actions in π' that belong to the same layer. Formally, if $\pi' = < \alpha_1; \ldots; \alpha_n >$ is the canonical linearization of $(\mathcal{A}_\pi, Ab(\pi))$ then the canonical parallelization π'' of $(\mathcal{A}_\pi, Ab(\pi))$ can be defined as follows: $\pi'' = < \{\alpha_1; \ldots; \alpha_{l_0}\}; \{\alpha_{l_0+1}; \ldots; \alpha_{l_1}\}; \ldots; \{\alpha_{l_{m-1}}; \ldots; \alpha_n\} >$ where

$l_0 \quad = min\{j \mid (\alpha_j, \alpha_{j+1}) \text{ are in different layers}\}$
$l_{i+1} = min\{j \mid j > l_i \wedge (\alpha_j, \alpha_{j+1}) \text{ are in different layers}\}$

Figure 3: The primed dependency graph for the parallel plans programs

It is worth observing that the canonical parallelization π'' of $(\mathcal{A}_\pi, Ab(\pi))$ is a minimal multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. More precisely, if $\pi'' =< \Gamma_1; \ldots; \Gamma_m >$ then each Γ_{i+1} is the set of all nodes with maximal distance i from the source in the graph representing $(\mathcal{A}_\pi, Ab(\pi))$ (as described in Section 5.3). Therefore each canonical plan defines a minimal parallel plan in the equivalence class it represents.

Proposition 17 *Let π be a totally ordered plan. If π' is the canonical parallelization of $(\mathcal{A}_\pi, Ab(\pi))$ then π' is a minimal multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$.*

Proof. We first show that π' is a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. Suppose that $\pi' =< \Gamma_1; \ldots; \Gamma_m >$ is not a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. $\exists \alpha, \beta, i, j : \alpha \prec \beta \in Ab(\pi) \wedge \alpha \in \Gamma_i \wedge \beta \in \Gamma_j \wedge i \geq j$. By definition of canonical plan, we have that if $i = j$ then $layer(\alpha) = layer(\beta)$ while if $i > j$ then $layer(\alpha) > layer(\beta)$, and in either case we obtain a contradiction. We now show that π' is a minimal multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$. If $\pi' =< \Gamma_1; \ldots; \Gamma_m >$ is a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$ then m is the maximal distance of all maximal distances of every node in the graph. Suppose that there exists a multi-level sort $\pi'' =< \Delta_1; \ldots; \Delta_p >$ of $(\mathcal{A}_\pi, Ab(\pi))$, with $p < m$. Consider an action $\alpha \in \Gamma_m$, and let Δ_i s.t. $\alpha \in \Delta_i$. By definition of canonical plan $\exists \beta_1, \ldots, \beta_{m-1} : \beta_1 \prec \beta_2 \in Ab(\pi) \wedge \beta_2 \prec \beta_3 \in Ab(\pi) \wedge \ldots \wedge \beta_{m-1} \prec \alpha \in Ab(\pi)$. Then in order for

28

π'' to be a multi-level sort of $(\mathcal{A}_\pi, Ab(\pi))$ there should exists $(m-1)$ sets of actions $\Delta_{j_1}, \ldots, \Delta_{j_{m-1}}$ s.t. $j_1 < j_2 < \ldots < j_{m-1} < i$ and $\beta_h \in \Delta_{j_h}$, for all $h \in [1, m-1]$. But this is not possible since $p < m$. $\qquad \square$

6 Conclusion

The limited scope of this paper prevents us from including a detailed discussion of all previous work on logic-based planning. However we will briefly mention that there are two major approaches: one models planning directly in logic, and the other does so indirectly using an action language.

The direct approach was started by Cordell Green [14, 15], who used a state argument on predicates, and rules (as we do in our approach). While Green's approach has clear intuitive appeal, it suffers from computational inefficiency in supporting the frame axioms. Thus, several techniques have been proposed to improve on Green's approach, including Kowalski's [19] meta-predicate `Holds`, and situation calculus by McCarthy [23], Pinto and Reiter [25], and others.

Our approach realizes a model very close to that originally proposed by Green without using any special construct; in fact, we only employ constructs supported in a running deductive database system $\mathcal{LDL}{++}$ [36].

A more recent line of work takes an indirect approach by using action description languages, such as $\mathcal{A}$ [11]. These languages are used to express planning and other problems, such as the Yale shooting problem [17]. Action languages provide an appealing representation of the planning problem. Moreover, their semantics can be formalized using $\mathcal{LDL}{++}$ choice programs and using techniques similar to those we used for planning — as outlined in the Appendix B. Thus, action languages reinforce this paper's thesis: Recent deductive database advances bring database and A.I. much closer together. Similar benefits can be claimed for transaction logic [4], which however introduces a new and special framework for updates, rather than relying on the standard Datalog rules used for expressing transitive closure queries in database textbooks and in a number of deductive database systems.

Acknowledgements

References

[1] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[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] C. Baral. Relating Logic Programming Theories of Actions and Partial Order Planning, *Annals of Math and AI*, vol. 21 (1997) Nos 2-4.

[4] A.J. Bonner and M. Kifer. An Overview of Transaction Logic. Theoretical Computer Science (TCS), 133:205-265, 1994.

[5] J. Chomicki. Polynomial-time computable queries in temporal deductive databases. In *PODS'90*, 1990.

[6] 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.

[7] 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.

[8] K. Erol, D.S. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 1995.

[9] 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.

[10] 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.

[11] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic programming*, 17:301–322, 1993.

[12] M.P. Georgeff. A theory of actions for multiagent planning. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 121–125. The MIT Press, 1984.

[13] S. Greco and C. Zaniolo, Greedy Algorithms in Datalog with Choice and Negation, In *Procs. 1998 Joint International Conference and Symposium on Logic Programming, JCSLP'98*, pp. 294-309, MIT Press, 1998.

[14] C.C. Green. Theorem Proving by Resolution as a Basis for Question-Answering Systems, *Machine Intelligence 4,* (eds. B. Meltzer and D. Michie), Edinburgh University Press. 1969.

[15] C.C. Green. Application of Theorem Proving to Problem Solving, Proc. 1969 Intl. Joint Conf. on Artificial Intelligence, pps 219–240. 1969.

[16] G. Grobe and R. Waldinger. Towards a theory of simultaneous actions. In J. Hertzberg, editor, *Proceedings of European Workshop on Planning*, number 522 in LNAI, pages 78–87. Springer-Verlag, 1991.

[17] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[18] A. Horz. On the relation of classical and temporal planning. In *Proc. of the Spring Symposium on Foundations of Automated Planning*, 1993.

[19] R.A. Kowalski. *Logic for Problem Solving*, North Holland Elsevier, 1974.

[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 circumscription to formalising common sense reasoning. *Artificial Intelligence*, 26:89–116, 1986.

[24] E. Pednault. Solving multiagent dynamic world problems in the classical planning framework. In *Reasoning about actions and plans: Proceedings of the 1986 Workshop*, pages 42–82. Morgan Kaufmann, 1986.

[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] P. Regnier and B. Fade. Complete determination of parallel actions and temporal optimization in linear plans of action. In J. Hertzberg, editor, *Proceedings of European Workshop on Planning*, pages 100–111. Springer-Verlag, 1991.

[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] D. Sacca' and C. Zaniolo, Deterministic and Non-Deterministic Stable Models, Journal of Logic and Computation (JLC), 1997.

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

[30] 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.

[31] H. Wang, Carlo Zaniolo: Nonmonotonic Reasoning in LDL++. In Logic-Based Artificial Intelligence (J. Minker Editor), Kluwer Academic Publishers, pp. 523-542, 2000.

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

[33] 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.

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

[35] 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.

[36] Zaniolo, C. et al. (1998) $\mathcal{LDL}++$ Documentation and Web Demo, 1988: http://www.cs.ucla.edu/ldl

[37] C. Zaniolo, S. Ceri, C. Faloutsos,R. Snodgrass, VS. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann Publisher, San Francisco, 1997.

A   Formal Semantics

If P is an instantiated program and M an interpretation for P, let $ST(M, ground(P))$ denote the positive program constructed from $ground(P)$ by the stability transformation defined by the interpretation M. Therefore, M is a stable model for P iff M is the least model for $ST(M, ground(P))$.

We can now prove our Theorem 1.

Theorem 1   *Let P be a logic program and Σ_P be a topological stratification for P in n strata. Then, M is a stable model for P iff for every $0 \leq j < n$*

$$M^{\leq j} = \{x \in M | stratum(x) \leq j\}$$

is a stable model for $ground^{\leq j}(P)$.

Proof. The "if" part of the proof is trivial. To prove the only-if part, observe that $M^{\leq j}$ satisfies all the rules in $ground^{\leq j}(P)$, thus it is a model for this program. By contradiction, assume that $M^{\leq j}$ is not a stable model for $ground^{\leq j}(P)$. Then, the least model of $ST(M^{\leq j}, ground^{\leq j}(P))$ is a proper subset of $M^{\leq j}$: let a be an atom that is in $M^{\leq j}$ but not in the least model of $ST(M^{\leq j}, ground^{\leq j}(P))$. Now, the set of rules in $ST(M, ground^{\leq j}(P))$, is a subset of those in $ST(M^{\leq j}, ground^{\leq j}(P))$; thus the least model of the latter cannot contain a either. But, the remaining rules in $ST(M, ground(P))$ have heads in strata higher than j, so they cannot produce a. Thus M fails the stability test, and M is not a stable model. This completes our proof by contradiction. $\square$

Therefore, Theorem 1 relates the stable models for a program to he stable models for its topological strata. The next theorem defines the relation between one stratum and the next.

Theorem 2   *Let P be a logic program with topological stratification Σ_P containing n strata. Then, for every $0 \leq j < n$:*

1. *If M is a stable model for $ground^{\leq j}(P)$, then every stable model for $\phi(M) \cup ground^{j+1}(P)$ is a stable model for $ground^{\leq j+1}(P)$, and*

2. *If N is a stable model for $ground^{\leq j+1}(P)$ then $N^{\leq j} = \{x \in N | stratum(x) \leq j\}$ is a stable model for $ground^{\leq j}(P)$.*

Proof.   To prove 1, say that N is the least model for the positive program $ST(N, \phi(M) \cup ground^{j+1}(P))$. Now, N can be computed by (i) deriving M directly from the facts in $\phi(M)$, and (ii) then deriving $N - M$ using the rules in $ST(N, ground^{j+1}(P))$. For our proof, we must show that N is the least model for $ST(N, ground^{\leq j}(P) \cup ground^{j+1}(P))$. Indeed, we have that: (i) the rules in $ST(N, ground^{\leq j}(P)) = ST(M, ground^{\leq j}(P))$ produce M, and (ii) the rules in $ST(N, ground^{j+1}(P))$ produce $N - M$.

To prove point 2, observe that N is also a least model for $Q = ST(N, ground^{\leq j}(P))$. Let $N^{\leq j}$ denote the atoms in N up to the j stratum; given that all the heads and goals in $ground^{\leq j}(P)$ are in $N^{\leq j}$, $N^{\leq j}$ is also a model for Q. Moreover, if $N^{\leq j}$ is not the least model for Q, N cannot be the least model for $ST(N, ground^{\leq j+1}(P))$ either. Thus, $N^{\leq j}$ is the least model for $Q = ST(N^{\leq j}, ground^{\leq j}(P)) = ST(N, ground^{\leq j}(P))$. This completes our proof. $\square$

Lemma 3 *Let P be a stratified choice program. Then $foe(P)$ has one or more stable model.*
Proof. Let us assign each `chosen_r` and `diffchoice_r` predicate generated by the expansion of a chosen rule r to the same stratum as the head of r. Thus the only negated goals in current stratum are the $\neg$`diffchoice_r` goals in the rules defining `chosen_r`. Thus, within each stratum we have the same situation as positive choice programs, which always have a stable model. $\square$

Consider now the temporal term $0 + 3$ (a short hand for $0 + 1 + 1 + 1$). We will say that this term has temporal rank 3. We can extend the notion of temporal rank to arbitrary ground terms, whereby, e.g., $f(a) + 1 + 1 + 1$ has temporal rank 3. By induction, therefore: (i) a term that does not unify with $X + 1$ has temporal rank zero, and (ii) a term that unifies with $X + 1$ has temporal rank $j + 1$, where j is the temporal rank of X.

Now, say that rule r' is obtained from a rule $r \in P$ by instantiating only r's temporal variable, if one exists with terms from P's Herbrand universe (the temporal variable of a rule r is the variable in the temporal argument of the head of r). Then r' will be called a *temporal snapshot* of r. If the temporal argument of the head of r contains no variable, then r and its temporal snapshot coincide. Otherwise, there are as many snapshots for r as there are terms in P's Herbrand Universe. The temporal rank of a snapshot is the temporal rank of its head. Let $Snap^j(r)$ denote the set of temporal snapshots of r with temporal rank j. Then,

$$Snap^j(P) = \bigcup_{r \in P} Snap^j(r)$$

denotes the set of *snapshots of P with temporal rank j*.
We have that:

$$ground(Snap^j(P)) = ground^j(P)$$

The following lemma follows immediately from the definitions:

Lemma 18 *Let P' denote the primed version of an XY-stratified choice program. Then, P' is a stratified choice program.*

Consider now an XY-stratified program Q. Then, $ground(foe(Q))$ can be topologically stratified by assigning every atom x in the Herbrand's base of $ground(foe(Q)$ to the stratum j where $j = rank(x)$. We can now prove Theorem 9

Theorem 9 *Every XY-stratified choice program has one or more stable models.*
Proof. Let Q be an XY-stratified choice program. Following Theorem 2, we can reason by induction: we assume that $ground^{\leq j}(foe(Q))$, for some $j \geq 0$, has as stable model M, and prove that $\phi(M) \cup ground^{j+1}(foe(Q))$ has a stable model.

Indeed, $G = \phi(M) \cup ground^{j+1}(foe(Q))$ is the ground instance of $\phi(M) \cup (Snap^{j+1}(foe(Q))$. Now let G' be the program obtained from G by priming the atoms having temporal rank $j + 1$. Obviously, G has a stable model iff G' does. If Q' is the primed version of Q then $\phi(M) \cup foe(Q')$ is a stratified choice program, and thus has a stable model and so does $\phi(M) \cup Snap^{j+1}(foe(Q'))$, and its ground version that coincides with G'.

By a similar argument, we can prove that $ground^0(foe(Q))$ has a stable model; this yields the base case that completes our inductive proof. $\square$

Since our planning program is XY-stratified, Theorem 4 follows as a corollary of this last theorem.

B   Action Languages

Several works have addressed the problem of modelling the logic of actions by means of logic programming languages. In particular, let us consider the language $\mathcal{A}$ proposed by Gelfond and Lifschitz for describing actions, who showed that it can be translated into a logic programming language which uses both classical negation and negation as failure [11]. The translation is proved to be sound and is used for temporal projection problems with incomplete information as well as for reasoning about the past. A description of an action domain in $\mathcal{A}$ consists of two kinds of propositions. Value propositions have the form: "F after $A_1; \ldots; A_m$" or "initially F", while effect propositions have the form: "A causes F if $P_1, \ldots, P_n$" or "A causes F" — where A, $A_1, \ldots, A_m$ are action names, and $F, P_1, \ldots, P_n$ are fluent expressions, i.e., fluent names possibly preceded by $\neg$.

It is worth analyzing whether, and how, our formalization of planning can be applied to the language $\mathcal{A}$. Indeed, a description of an action domain in $\mathcal{A}$ can be translated into a choice logic program as follows. For each value proposition of the form: "initially F" we introduce an exit rule:

$$\texttt{F}^{\texttt{0}}.$$

Action names are represented by a database relation of the form:

$$\texttt{action(A)}.$$

Each effect proposition "A causes F if $P_1, \ldots, P_n$" is translated into one of the rules:

$$\texttt{add}^{\texttt{J}}(\texttt{F}) \quad \leftarrow \quad \texttt{fired}^{\texttt{J}}(\texttt{A}), \texttt{P}_\texttt{1}^\texttt{J}, \ldots, \texttt{P}_\texttt{n}^\texttt{J}.$$
$$\texttt{del}^{\texttt{J}}(\texttt{F}) \quad \leftarrow \quad \texttt{fired}^{\texttt{J}}(\texttt{A}), \texttt{P}_\texttt{1}^\texttt{J}, \ldots, \texttt{P}_\texttt{n}^\texttt{J}.$$

depending on whether F is a fluent name, or a fluent name preceded by $\neg$. Action selection is simply modeled by the rule:

$$\texttt{fired}^{\texttt{J}}(\texttt{A}) \quad \leftarrow \quad \texttt{action(A)}, \texttt{choice}^{\texttt{J}}(\texttt{A}).$$

Finally, state changes are modeled by the rules (6), (7), (8) and (9), introduced in Section 3. Following Corollary 8 of Section 4, the validity of value propositions of the form "F after $A_1; \ldots; A_m$" can be then checked by including the clauses:

$$\texttt{chosen}^{\texttt{0}}(\texttt{A}_\texttt{1}).$$
$$\texttt{chosen}^{\texttt{1}}(\texttt{A}_\texttt{2}).$$
$$\ldots$$
$$\texttt{chosen}^{\texttt{m}-\texttt{1}}(\texttt{A}_\texttt{m}).$$

into the program $foe(\mathcal{CH}(\Pi))$, and by checking whether the F is true in the stable model of the resulting logic program.

The above discussion illustrates how a description of an action domain in $\mathcal{A}$ can be translated into a choice logic program. It is worth comparing this formalization of $\mathcal{A}$ with the formalization described in [11]. On the one hand, our setting supports the actual process of plan generation given an initial state, and it does not require a given sequence of actions to reason about the effects of their execution. This is due to one of the key features of our formalization, that is, the ability of expressing

the choice of one action to be executed. Moreover, our translation extends also to "non-similar" effect propositions [11], or propositions of the form: "A causes $F_1, F_2, \ldots, F_h$ if $P_1, \ldots, P_n$". On the other hand, while Gelfond and Lifschitz's translation supports temporal reasoning with incomplete information, our formalization interprets the description of the initial state, and of states in general, as complete (closed world assumption).

The language $\mathcal{A}$, by itself, has no state-variables in its *syntax* (though its *semantics* is state-based), and this does not explicitly fall within the realm of the situation calculus. As Gelfond and Lifschitz's [11, 3] translation takes any action program and converts it into an equivalent logic program, the operational mechanisms of logic programming, as well as implementations of logic programming may be used, in principle, to implement reasoning about actions in $\mathcal{A}$. The logic programs produced by this translation use Kowalski's *Holds* predicate[3], but also include nonmonotonic models of negation. For example, Baral's transformation[3] produces logic programs that are *acyclic*[2] ad hence have some nice computation properties.

As we have shown in the above discussion, our language allows us to represent a large fragment of Gelfond and Lifschitz's $\mathcal{A}$ language, within our polynomial XY-stratified programs. However we do not represent incomplete information about states, which Gelfond and Lifschitz can do. Conversely, our framework is shown to solve parallel and partial order planning problems, while they do not do so.