Relating Stable Models and AI Planning Domains

V.S. Subrahmanian
University of Maryland
College Park, MD 20742, U.S.A.

Carlo Zaniolo
University of California
Los Angeles, CA 90024, USA.

Abstract

In this paper, we show that there is a simple connection between logic programming and planning. The main result of this paper is the following: given any planning domain consisting of an initial state, and a set of operation definitions, this domain can be translated, in linear-time, to a logic program such that a given goal G is achievable in the planning domain iff a related goal $G^\star$ is true in some stable model of the logic program obtained by the translation. We show that this translation yields at least two interesting consequences: (1) methods to update databases can be used to handle surprises when executing plans (i.e. a surprise occurs when an initial plan is partly executed, but one of the resulting intermediate states differs, perhaps due to external reasons, from what is predicted). (2) rigid actions, which are actions that *must* be executed when their pre-conditions are true, can be easily accommodated within our framework as well.

1 Introduction

One of the fundamental tasks of artificial intelligence research is to allow an autonomous reasoning agent, such as a robot, to achieve goals by creating sequences of operations (or plans) based on a set of primitive operations that the agent can execute. Each of these primitive operations has three aspects to it – a set of preconditions under which it is possible to execute the operation, a set of facts ("add" list) that become true after the operation is executed, and a set of facts ("delete" list) that cease to be true after the operation is executed.

Expressing such a specification of an operation using formal logic has been studied widely. The situation calculus, due to McCarthy [8] is perhaps the best known such formalism. A resurgence of interest in the situation calculus, and its implementation via logic programming has recently been studied by Pinto and Reiter [9]. At the same time, Gelfond and Lifschitz [6] have argued that incorporating actions into logic programming may be achieved by using two forms of negation – explicit negation, and nonmonotonic negation. Dung [3] has argued that there are various inadequacies in the Gelfond-Lifschitz framework. He has suggested a new language that incorporates actions into logic programming, studied the semantics of this language, and proven various elegant soundness and completeness results.

Erol, Nau and Subrahmanian[4] have shown that there is a very simple and natural translation of planning domains (whose operations have empty delete lists) to negation-free logic programs such that a goal G can be achieved from the planning domain iff a related goal can be proved from the negation-free logic program obtained by the above-mentioned translation. A key point to note is that in deletion-free planning domains, execution of operations leads to monotonicity – nothing that was true initially ever becomes false as a result of the execution of operations. However, most operations modify the world, making certain "new" facts true, and falsifying others.

In this paper, we show that given any planning domain (including one with delete lists), there is a uniform procedure that converts the planning domain into a "choice" logic program. In a choice logic program, nondeterministic choices can be made. The rough analogy with planning is that when several operations can be executed in a given state, then a nondeterministic choice of one of them must be made. Choice logic programs can be transformed into logic programs with negation.

Three contributions are made in this paper:

- **Planning via Stable Models:** A given goal G is achievable with respect to the planning domain iff a related goal $G^{\star}$ is true in some stable model of the logic program (with negation) constructed using the framework just mentioned.
- **Handling Surprises:** When surprises occur and there is a discrepancy between the state we expect (based on a specification of the initial state, and the actions executed upto this point in time), then these discrepancies can be treated as updates to the logic program constructed above.
- **Rigid Actions:** Planning theory mandates that if a set of actions are executable in a given state, then one amongst these actions is chosen for execution. However, in many situations (particularly those related to safety devices in intelligent plant monitoring and control systems), there are certain actions that *must* be executed when they are enabled. We show how such actions can be incorporated within our framework.

The idea that there is a connection between planning and logic programming is not new. Bonner and Kifer [1] proposed a formalism called *transaction logic programming* where they developed a path-based modal semantics for logic programming and show how planning operators can be incorporated in their language. In contrast, we do not extend logic programming in any way.

The stable semantics for logic programs is one of the best known semantics for logic programming with negation. We present a uniform translation of planning domains into logic programs whose stable models capture the set of goals achievable from the planning domain. The LDL system[2] that has been implemented over the years at MCC has facilities to support the use of the choice construct, and hence, LDL can be used as a platform upon which to develop efficient implementations of planning systems based upon the translation of planning domains to logic programs described herein.

2 Preliminaries

Definition 2.1 Let $\mathcal{L}$ be any first-order language generated by finitely many constant symbols, predicate symbols, and function symbols. Then a *state* is any finite set of ground atoms in $\mathcal{L}$.[1]

Intuitively, a state tells us which ground atoms are currently true: if a ground atom A is in state S, then A is true in state S, and if $B \notin S$, then B is false in state S. Thus, a state is simply an Herbrand interpretation for the language $\mathcal{L}$, and hence each formula of first-order logic is either satisfied or not satisfied in S according to the usual first-order logic definition of satisfaction.

Definition 2.2 Let $\mathcal{L}$ be an ordinary first-order language. Then a *planning operator* α is a 4-tuple $(\texttt{Name}(\alpha), \text{Pre}(\alpha), \text{Add}(\alpha), \text{Del}(\alpha))$, where

1. $\texttt{Name}(\alpha)$ is a syntactic expression of the form $\alpha(X_1, \ldots, X_n)$ where each X_i is a variable symbol of $\mathcal{L}$;
2. $\text{Pre}(\alpha)$ is a finite set of atoms, called the *precondition list* of α, whose variables are all from the set $\{X_1, \ldots, X_n\}$;
3. $\text{Add}(\alpha)$ and $\text{Del}(\alpha)$ are both finite sets of atoms (possibly non-ground) whose variables are taken from the set $\{X_1, \ldots, X_n\}$. $\text{Add}(\alpha)$ is called the *add list* of α, and $\text{Del}(\alpha)$ is called the *delete list* of α.

Observe that negated atoms are not allowed.

Definition 2.3 A *first-order planning domain* (or simply a *planning domain*) is a pair $\mathbf{P} = (S_0, \mathcal{O})$, where S_0 is a state called the *initial state*, and $\mathcal{O}$ is a finite set of planning operators. The *language* of $\mathbf{P}$ is the first-order language $\mathcal{L}$ generated by the constant, function, predicate, and variable symbols appearing in $\mathbf{P}$, along with an infinite number of additional variable symbols.

[1] It is standard practice to assume that first-order languages contain only finitely many constant symbols, and that states contain only finitely many atoms.

Definition 2.4 A *goal* is a conjunction of atoms which is existentially closed (i.e., the variables, if any, are existentially quantified). (We will often abuse notation and write a goal as a conjunction of atoms without writing the quantifiers.)

Definition 2.5 A *planning problem* is a triple $\mathbf{PP} = (S_0, \mathcal{O}, G)$, where $(S_0, \mathcal{O})$ is a planning domain and G is a goal.

Definition 2.6 Let $\mathbf{P} = (S_0, \mathcal{O})$ be a planning domain, α be an operator in $\mathcal{O}$ whose name is $\alpha(X_1, \ldots, X_n)$, and θ be a substitution that assigns ground terms to each $X_i, 1 \leq i \leq n$. Suppose that the following conditions hold for states S and S': (1) $\{A\theta : A \text{ is an atom in } \mathrm{Pre}(\alpha)\} \subseteq S$ and (2) $S' = (S - (\mathrm{Del}(\alpha)\theta)) \cup (\mathrm{Add}(\alpha)\theta)$. Then we say that α is *θ-executable* in state S, *resulting* in state S'. This is denoted symbolically as $S \stackrel{\alpha,\theta}{\Longrightarrow} S'$.

Definition 2.7 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain and G is a goal. A *plan that achieves* G is a sequence $S_0, \ldots, S_n$ of states, a sequence $\alpha_1, \ldots, \alpha_n$ of planning operators, and a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that $S_0 \stackrel{\alpha_1,\theta_1}{\Longrightarrow} S_1 \stackrel{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \stackrel{\alpha_n,\theta_n}{\Longrightarrow} S_n$ and G is satisfied by S_n, i.e. there exists a ground instance of G that is true in S_n. The *length* of the above plan is n.

3 From Planning to Active Databases

Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain. In this section, we will develop a connection between the stable model semantics for logic programs, active databases, and AI planning domains.

3.1 Choice Models and AI Planning Domains

In this section, we will show how, given a planning domain, $\mathbf{P}$, we can associate a logic program, $\mathcal{CH}(\mathbf{P})$, with choice constructs (due to Sacca and Zaniolo [11]).

For every ground atom $\mathtt{p}(\vec{\mathtt{t}}) \in S_0$, $\mathcal{CH}(\mathbf{P})$ contains the unit clause $\mathtt{p}(0, \vec{\mathtt{t}}) \leftarrow$. Intuitively, this clause says that $\mathtt{p}(\vec{\mathtt{t}})$ is true at stage 0. For every planning operator $\alpha \in \mathcal{O}$, we have the following rules:

$$\mathtt{firable}(\mathtt{J}, \mathtt{Name}(\alpha)) \quad \leftarrow \quad \mathtt{p}_1(\mathtt{J}, \vec{\mathtt{t}}_1) \,\&\, \ldots \,\&\, \mathtt{p}_n(\mathtt{J}, \vec{\mathtt{t}}_n) \qquad (1)$$

where $\mathrm{Pre}(\alpha) = \{\mathtt{p}_1(\vec{\mathtt{t}}_1), \ldots, \mathtt{p}_n(\vec{\mathtt{t}}_n).\}$. All this says is that the action α is firable (with suitable instantiation of the variables in its name) at stage $\mathtt{J}$ just in case each of its pre-conditions is true at stage $\mathtt{J}$. Furthermore, if $\mathtt{p}(\vec{\mathtt{t}})$ is in $\mathrm{Add}(\alpha)$, then we have the rule:

$$\mathtt{add}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}})) \quad \leftarrow \quad \mathtt{fired}(\mathtt{J}, \mathtt{Name}(\alpha)). \qquad (2)$$

At this point, a brief side note is in order. When a k-ary predicate symbol $\mathtt{p}$ occurs in the planning domain $\mathbf{P}$, we use $\mathtt{p}$ to denote two symbols in the language of $\mathcal{CH}(\mathbf{P})$ – (1) a $(k+1)$-ary predicate symbol, and (2) a k-ary function symbol. Strictly speaking, we should use two different symbols, but we will abuse notation – the reader can easily determine from context whether an occurrence of $\mathtt{p}$ in $\mathcal{CH}(\mathbf{P})$ denotes a function symbol, or whether it denotes a predicate symbol.

Similarly to the above, if $\mathtt{p}(\vec{\mathtt{t}})$ is in $\mathrm{Del}(\alpha)$, then we have the rule:

$$\mathtt{del}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}})) \leftarrow \mathtt{fired}(\mathtt{J}, \mathtt{Name}(\alpha)). \tag{3}$$

In addition, we need a rule saying that at a given point in time, one, and only one firable action is selected for execution. This is achieved by using the choice operator and writing the rule:

$$\mathtt{fired}(\mathtt{J}, \mathtt{Name}(\alpha)) \leftarrow \mathtt{firable}(\mathtt{J}, \mathtt{Name}(\alpha)) \,\&\, \mathtt{choice}(\mathtt{J}, \mathtt{Name}(\alpha)). \tag{4}$$

Intuitively, the above rule says that the (appropriate instance of the) action α is fired if (the appropriate instance of) it is firable and if the nondeterministic `choice` construct picks this instance. The predicate `choice` is a special predicate that, for each instance of its first argument, nondeterministically picks a unique instance of the second argument – hence the word `choice`. The semantics of such choice constructs have been extensively studied in the database literature [2, 11], and are fully implemented in the LDL system fielded by MCC [2].

Frame Axioms. In addition, for each predicate $\mathtt{p}$, we have the following rules in $\mathcal{CH}(\mathbf{P})$:

$$\mathtt{p}(\mathtt{J}+\mathtt{1}, \vec{\mathtt{t}}) \leftarrow \mathtt{add}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}})). \tag{5}$$

$$\mathtt{p}(\mathtt{J}+\mathtt{1}, \vec{\mathtt{t}}) \leftarrow \mathtt{p}(\mathtt{J}, \vec{\mathtt{t}}) \,\&\, \mathbf{not}(\mathtt{del}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}}))). \tag{6}$$

The above two rules play a role akin to those of "frame" axioms; they specify that if a new fact is added (resp. deleted) at stage $\mathtt{J}$, then this fact becomes true (resp. false) at stage $\mathtt{J}+\mathtt{1}$.

Given a planning domain $\mathbf{P} = (S_0, \mathcal{O})$, we can convert $\mathbf{P}$ into a choice logic program, $\mathcal{CH}(\mathbf{P})$, by associating, with each operator $\alpha \in \mathcal{O}$, the clauses (1)-(4) above, and for each predicate symbol $\mathtt{p}$, the clauses (5) and (6) above.

3.2 From Choice Programs to Choice-Free Programs

Using the techniques specified in [11], any choice logic program can be converted into a logic program with negation. In particular, clause (4) can be converted to the following set of clauses:

$$\mathtt{fired}(\mathtt{J}, \mathtt{Name}(\alpha)) \leftarrow \mathtt{firable}(\mathtt{J}, \mathtt{Name}(\alpha)) \,\&\, \mathtt{chosen}(\mathtt{J}, \mathtt{Name}(\alpha)). \tag{7}$$

$$\begin{aligned} \mathtt{chosen}(\mathtt{J}, \mathtt{Name}(\alpha)) \leftarrow\ & \mathtt{firable}(\mathtt{J}, \mathtt{Name}(\alpha)) \,\&\, \\ & \mathbf{not}\,(\mathtt{diffchoice}(\mathtt{J}, \mathtt{Name}(\alpha))). \end{aligned} \tag{8}$$

$$\mathtt{diffchoice}(\mathtt{J}, \mathtt{Name}(\alpha)) \leftarrow \mathtt{chosen}(\mathtt{J}, \mathtt{Name}(\alpha_1)) \,\&\, \mathtt{Name}(\alpha) \neq \mathtt{Name}(\alpha_1). \tag{9}$$

We use the notation $\mathcal{LP}(\mathbf{P})$ to denote the logic program (without choice) obtained by translating each operator $\alpha \in \mathcal{O}$ into clauses (1),(2),(3),(4), and associating, with each predicate symbol in our original vocabulary (i.e. the vocabulary of $\mathbf{P}$), the clauses (5) and (6). In addition, the clauses (7), (8) and (9) are in $\mathcal{LP}(\mathbf{P})$.

Given $\mathcal{LP}(\mathbf{P})$, we use $\mathsf{DET}(\mathbf{P})$ to denote the logic program obtained by:

1. removing, from $\mathcal{LP}(\mathbf{P})$, all rules with `diffchoice` as the predicate symbol in the head of the clause, and
2. removing all positive and negative literals of the form `diffchoice`$(-,-)$ from the body of the remaining rules.

$\mathsf{DET}(\mathbf{P})$ will be called the *deterministic version* of $\mathbf{P}$.

Definition 3.1 Let Q be any logic program with negated goals and choice goals. Q is said to be *locally stratified modulo choice* iff its deterministic version, $\mathsf{DET}(Q)$, is locally stratified.

Proposition 3.1 For any planning domain, $\mathbf{P}$, $\mathsf{DET}(\mathbf{P})$ is locally stratified; hence, $\mathcal{CH}(\mathbf{P})$, is locally stratified modulo choice. $\square$

Thus, given a planning domain $\mathbf{P}$, we can convert $\mathbf{P}$ into a logic program $\mathcal{LP}(\mathbf{P})$ by first converting $\mathbf{P}$ into a choice logic program, and then converting this choice logic program into a (choice-free) logic program that may contain negation in its clauses.

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

Iterated Fixpoint

basis: $M^0 := \emptyset$

induction: for $0 < j$
$M^j :=$ the unique perfect model (cf. Przymusinski [10]) of $M_{j-1} \cup$ $ground^j(Q)$,

where: $ground^j(Q)$ denotes the rules of $ground(Q)$ with head in the j-th stratum[2].

The result, of the iterated fixpoint procedure just described, $M^\omega = \cup_{0 \leq j < \omega}$ M^j, is equal to the perfect model, and unique stable model, of Q [10].

Suppose now that $\mathbf{P}$ is a planning domain – hence, by Proposition 3.1, $\mathcal{CH}(\mathbf{P})$ is locally stratified modulo choice. Observe that $\mathcal{LP}(\mathbf{P})$ can be split into the set of rules defining the `diffchoice` predicate, denoted $\mathsf{DFC}(\mathcal{LP}(\mathbf{P}))$, and the remaining rules $\mathcal{LP}(\mathbf{P}) - \mathsf{DFC}(\mathcal{LP}(\mathbf{P}))$. Then the rules in $ground(\mathcal{LP}$

[2]When function symbols are present, this iteration may go on for transfinitely long.

$(\mathbf{P}) - \mathsf{DFC}(\mathcal{LP}(\mathbf{P})))$ can be partitioned according to the strata of $\mathsf{DET}(\mathbf{P})$ to which their head belongs: again let $ground^j(\mathbf{P})$ denote the set of rules in $ground(\mathcal{LP}(\mathbf{P}))$ whose head belong to the j-th stratum of $\mathsf{DET}(P)$. In the presence of choice, the iterated fixpoint computation can be extended as follows:

Iterated Choice Fixpoint

basis: $M^0 := \emptyset$

induction: for $0 < j$
$M^j :=$ a stable model of $M^{j-1} \cup ground^j(\mathcal{LP}(\mathbf{P})) \cup \mathsf{DFC}(\mathcal{LP}(\mathbf{P}))$.

Since at each step there could be several stable models (each corresponding to a choice model for that stratum), we have a non-deterministic computation. Each result, of such a computation, i.e., $M^\omega = \bigcup_{0 \leq j < \omega} M^j$, will be called *a result of the Iterated Choice Fixpoint on P*.

Theorem 3.1 (Sacca and Zaniolo [11]) Any logic program that is locally stratified modulo choice has at least one stable model. $\square$

It follows immediately from Theorem 3.1 and by Proposition 3.1 that:

Proposition 3.2 Suppose $\mathbf{P}$ is a planning domain. Then: $\mathcal{LP}(\mathbf{P})$ has at least one stable model. $\square$

The following result shows an elementary property of the stable models of $\mathcal{LP}(\mathbf{P})$.

Theorem 3.2 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain. Let M be any stable model of $\mathcal{LP}(\mathbf{P})$. Then, for any integer $j \geq 0$, there is at most one atom in M of the form $\mathtt{fired(j, -)}$. $\square$

The above theorem is very useful because it shows that every stable model of $\mathcal{LP}(\mathbf{P})$ causes at most one action to be executed at any given point in time. The following theorem states that if M is a stable model, and α is *any* action firable at stage j, then there is a stable model M' that is exactly like M upto stage $(j-1)$ inclusive, but which causes action α to be fired at stage j (note that α may not be the same action that was fired at stage j in stable model M).

Theorem 3.3 Suppose M is a stable model of $\mathcal{LP}(\mathbf{P})$. Let $[M]_\mathtt{j} = \{\mathtt{q(j', \vec{t})} | \mathtt{q(j', \vec{t})} \in M$ and $\mathtt{j'} < \mathtt{j}\}$. Let α be any operation such that $\mathtt{firable(J, Name(}\alpha)\theta) \in M$, but $\mathtt{fired(J, Name(}\alpha)\theta) \notin M$. Then there is a stable model M' of $\mathcal{LP}(\mathbf{P})$ such that $[M]_\mathtt{j} = [M']_\mathtt{j}$ and such that $\mathtt{fired(J, Name(}\alpha)\theta) \in M'$. $\square$

It turns out that there is a close connection between the achievability of goals in the planning domain, and the truth of goals in the stable models of $\mathcal{LP}(\mathbf{P})$. The following result explicitly states and proves this connection.

Theorem 3.4 (Representing Achievability of Goals by Stable Models) Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain, and $G =$

$$\mathtt{p_1}(\vec{\mathtt{t}}_1)\,\&\ldots\&\,\mathtt{p_k}(\vec{\mathtt{t}}_\mathtt{k})$$

is a goal. Let $G^\star =$

$$(\exists \mathtt{J})\,(\mathtt{p_1}(\mathtt{J}, \vec{\mathtt{t}}_1)\,\&\ldots\&\,\mathtt{p_k}(\mathtt{J}, \vec{\mathtt{t}}_\mathtt{k})).$$

Then: G is achievable from $\mathbf{P}$ iff $G^\star$ is true in some stable model of $\mathcal{LP}(\mathbf{P})$. $\Box$

Proposition 3.3 (Complexity of Transformation) The complexity of computing $\mathcal{LP}(\mathbf{P})$, given as input, a planning domain $\mathbf{P} = (S_0, \mathcal{O})$ is linear-time. $\Box$

Theorem 3.4 shows that any planning domain can be converted into a logic program with nonmonotonic negation such that a goal G is achievable from the planning domain iff a related goal $G^\star$ is true is some stable model of the logic program. Furthermore, the conversion of the planning domain into such a logic program can be achieved in linear-time, and the transformation of G into $G^\star$ can be achieved in linear time as well. The following two sections present two areas in which theorem 3.4 can be used to contribute to the theory of planning.

4 Surprises

Existing theories of planning are unable to account for "surprises." In the real world, it is impossible to correctly capture what is actually true in the world. When the robot (or planning agent) discovers an error, it comes as a *surprise* to the robot, which must then re-plan so as to achieve the desired goal. In this section, we will argue that *surprises* are analogous to *updates.* The asymmetric nature of clauses (5) and (6) in $\mathcal{LP}(\mathbf{P})$ necessitates a slight modification of the above. In addion, rule (5) in $\mathcal{LP}(\mathbf{P})$ is replaced with the rule

$$\mathtt{p}(\mathtt{J}+1, \vec{\mathtt{t}}) \quad \leftarrow \quad \mathtt{add}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}}))\,\&\,\mathbf{not}\,(\mathtt{must\_del}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}}))). \qquad (10)$$

We will argue that the resulting logic program, denoted $\mathcal{LPSU}(\mathbf{P})$, "handles the surprise" in a way made precise in this section (cf. Theorem 4.1).

Definition 4.1 Suppose $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain. A *surprise*, $\mathcal{SU}$, is a set of atoms of the form $\mathtt{add}(\mathtt{j}, -), \mathtt{must\_del}(\mathtt{j}, -)$.

Alternatively, a surprise can be viewed as a function that takes as input, an integer $\mathtt{j} > \mathtt{0}$ and returns as output, a set, $\mathcal{SU}(\mathtt{j})$ of atoms (possibly empty) of the form $\mathtt{add}(\mathtt{j}, -)$ and $\mathtt{must\_del}(\mathtt{j}, -)$. We will often abuse notation and use both definitions when studying surprises.

Definition 4.2 A surprise $\mathcal{SU}$ is said to *start at step* $\mathtt{j} > \mathtt{0}$ iff $\mathtt{j}$ is the smallest integer such that $\mathcal{SU}(\mathtt{j}) \neq \emptyset$. Surprise $\mathcal{SU}$ is said to `end` *at step* $\mathtt{k} > \mathtt{0}$ iff $\mathcal{SU}(\mathtt{k}) \neq \emptyset$ and for all integers $\mathtt{m} > \mathtt{k}$, $\mathcal{SU}(\mathtt{m}) = \emptyset$.

Throughout this paper, we will assume that surprises always have an end, and that they do not occur over an infinite period of time.

Intuitively, if $\mathtt{add}(\mathtt{j}, \mathtt{A})$ is in surprise $\mathcal{SU}$, then this means that $\mathtt{A}$ is a fact that is found to be true (and whose truth may contradict earlier beliefs) at stage $\mathtt{j}$.

Surprises have a number of effects on planning, both *a priori*, static, plan generation, as well as on-the-fly planning. Two obvious effects are the following:

- **Anticipated Surprises:** When constructing a static plan, one may wish to *prepare* for surprises. This is a common aspect of human reasoning – a human planning to drive from point A to point B may be aware that road R *may* have a traffic jam (based on past experience, and s/he may have an alternative route in mind which s/he can "switch" to.
- **Plan Repair:** When *executing a plan*, a surprise may render the plan infeasible. In this case, a new plan must be dynamically recomputed (if possible). We will show how our framework can be used to generate such plans.

In this paper, we will consider plan repair – the (interesting) topic of planning for anticipated surprises is deferred to later work.

The intuition about how surprises are to be incorporated into planning is very simple: suppose that we are given a planning problem $\mathbf{PP} = (S_0, \mathcal{O}, G)$, where $(S_0, \mathcal{O})$ is a planning domain and G is a goal. At time $\mathtt{0}$, an initial plan

$$\mathcal{P}_0 \quad = \quad S_0^0 \stackrel{\alpha_1^0,\theta_1^0}{\Longrightarrow} S_1^0 \stackrel{\alpha_2^0,\theta_2^0}{\Longrightarrow} S_2^0 \cdots \stackrel{\alpha_n^0,\theta_n^0}{\Longrightarrow} S_n^0$$

is generated where $S_0^0 = S_0$. An action that takes S_0^0 is executed leading to the state S_1^0. Now certain surprises occur, i.e. certain atoms that were expected to be true in state S_1^0 are found to be false (these are atoms of the form $\mathtt{must\_del}(\mathtt{1}, \mathtt{A})$). In addition, certain atoms that were expected to be false turn out to be true, (these are atoms of the form $\mathtt{add}(\mathtt{1}, \mathtt{B})$). The original plan, $\mathcal{P}_0$ corresponded, by Theorem 3.4, to one stable model of

$\mathcal{LP}(\mathbf{P})$ – this is also a stable model of $\mathcal{LPSU}(\mathbf{P})$ because when surprises do not occur, the stable models of $\mathcal{LP}(\mathbf{P})$ and $\mathcal{LPSU}(\mathbf{P})$ coincide as there no clauses in $\mathcal{LPSU}(\mathbf{P})$ with a head of the $\texttt{must\_del}(-,-)$. Now, each atom in $\mathcal{SU}(\texttt{1})$ must be added as a fact to $\mathcal{LPSU}(\mathbf{P})$, and a new stable model of the expanded program must be computed.
The following definitions formalize how the occurrence of a surprise affects the existence/nonexistence of a plan.

Definition 4.3 Suppose $\mathbf{PP} = (S_0, \mathcal{O}, G)$ is a planning problem and $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain. Suppose $\mathcal{P}_0 =$

$$S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n$$

is a plan that achieves goal G, and suppose $\mathcal{SU}$ is a surprise that starts at step $\texttt{j} > \texttt{0}$. Goal G is said to be *achievable after* $\texttt{j} > \texttt{0}$ *steps of execution of plan* $\mathcal{P}_0$ iff G is achievable from the planning domain $(S', \mathcal{O})$ where $S' = (S_{\texttt{j}} - \{\texttt{A} \mid \texttt{must\_del}(\texttt{j},\texttt{A}) \in \mathcal{SU}\}) \cup \{\texttt{B} \mid \texttt{add}(\texttt{j},\texttt{B}) \in \mathcal{SU}\}$.

The following theorem says that surprises can just be viewed as updates in active databases.

Theorem 4.1 Suppose $\mathbf{PP} = (S_0, \mathcal{O}, G)$ is a planning problem and $\mathbf{P} = (S_0, \mathcal{O})$ is a planning domain. Suppose $\mathcal{P}_0 = S_0 \overset{\alpha_1,\theta_1}{\Longrightarrow} S_1 \overset{\alpha_2,\theta_2}{\Longrightarrow} S_2 \cdots \overset{\alpha_n,\theta_n}{\Longrightarrow} S_n$ is a plan that achieves goal G, and suppose $\mathcal{SU}$ is a surprise that starts at step $\texttt{j} > \texttt{0}$. Then: goal $G = (\exists)(\texttt{p}_1(\vec{\texttt{t}}_1) \,\&\, \ldots \,\&\, \texttt{p}_\texttt{k}(\vec{\texttt{t}}_\texttt{k}))$ is achievable after $\texttt{j} > \texttt{0}$ steps of execution of plan $\mathcal{P}_0$ iff the goal

$$G^{\flat} \quad = \quad (\exists \texttt{J})(\texttt{J} \geq \texttt{j} \,\&\, (\texttt{p}_1(\vec{\texttt{t}}_1) \,\&\, \ldots \,\&\, \texttt{p}_\texttt{k}(\vec{\texttt{t}}_\texttt{k}))$$

is true in some stable model $M \supseteq \{\texttt{fired}(\texttt{0}, \alpha_1\theta_1), \ldots, \texttt{fired}(\texttt{j}-\texttt{1}, \alpha_\texttt{j}\theta_\texttt{j})\}$ of the logic program $\mathcal{LPSU}(\mathbf{P}, \mathcal{SU}, \texttt{j}) = \mathcal{LPSU}(\mathbf{P}) \cup \{\texttt{add}(\texttt{j}-\texttt{1}, \texttt{p}(\vec{\texttt{t}})) \mid \texttt{add}(\texttt{p}(\vec{\texttt{t}})) \in \mathcal{SU}(\texttt{j})\} \cup \{\texttt{must\_del}(\texttt{j}-\texttt{1}, \texttt{p}(\vec{\texttt{t}})) \mid \texttt{must\_del}(\texttt{p}(\vec{\texttt{t}})) \in \mathcal{SU}(\texttt{j})\}$. $\Box$

Theorem 4.1 has the following important implication for plan repair: suppose an initial tentative plan $\mathcal{P}_0$ is generated with respect to the original statement of the initial state given by the user. When the plan is being executed, and an unexpected situation is encountered (e.g. robot sensors may be responsible for the observation that block $\texttt{e}$ is green, not red as originally specified), then theorem 4.1 says that all that needs to be done is to add the facts of the form $\texttt{add}(\texttt{j},-)$ and $\texttt{must\_del}(\texttt{j},-)$ generated by the surprise, and continue by incrementally generating a new stable model. Zaniolo [13] has described a procedure whereby this can be operationally accomplished on top of the LDL database system. In short, plan modification in the presence of surprises corresponds to incremental switching from one stable model to another. Furthermore, existing deductive database systems (such as LDL) already provide an efficient platform on which to execute such forms of plan modification.

5 Rigid Actions

One of the fundamental aspects of AI planning is *choice* – in a given state S, several operations may be individually executable; however, a choice is made, and one of these operations is selected and executed. It is precisely this notion of choice that provides a correspondence between logic programs with choice constructs (Sacca and Zaniolo [11]), and AI planning domains.

In many domains however, there may be a variety of operations that *must* be executed whenever their pre-conditions are satisfied. We term such operations *rigid* operations. Such operations may include safety devices in intelligent plant monitoring devices and mission-critical control systems. In many of these cases, these rigid actions can be viewed as "background" actions that do not mitigate the choice of execution of other (non-rigid) actions. Our logical formalism for choice-based planning, as exemplified by the construction of $\mathcal{LP}(\mathbf{P})$, can be used to support such rigid actions. Before showing how this is formally accomplished, we give a simple example below.

Example 5.1 (Plant Control) Let us suppose that we have an intelligent plant monitoring device whose task is to achieve a certain goal. We are not concerned with the actions that are needed to accomplish this goal. However, the device has a rigid action called $\mathsf{cool}(C, T)$ that reduces the current temperature, T, of component C by five degrees Celsius. Any component whose temperature exceeds a given threshold (say 100 degrees) must be brought back below this threshold by repeated applications of this "cooling" operator. The operator $\mathsf{cool}(C, T)$ can be described as follows:

- $\mathrm{Pre}(\mathsf{cool}(C, T)) = \{\mathtt{temp}(C, T), T > 100\}$.
- $\mathrm{Add}(\mathsf{cool}(C, T)) = \{\mathtt{temp}(C, T - 5)\}$.
- $\mathrm{Del}(\mathsf{cool}(C, T)) = \{\mathtt{temp}(C, T)\}$.

Note that the fact that a given component's temperature exceeds the threshold may, in fact, come as a surprise to the plant monitoring system, and so the temperature may have gone 20 or 30 degrees (for example) over the threshold before it is noticed by the sensors. Thus, the cooling operation must be repeatedly applied until the pre-condition of the rigid operations is no longer satisfied.

Let us suppose that certain actions $\rho_1, \ldots, \rho_k$ are designated as rigid actions. In general, two rules are said to "interfere" iff only one of them can actually be fired in any given state, even if both of them are firable in that state. In this paper, we will assume that rigid actions do not "interfere" with each other and/or with other non-rigid actions. Then we can incorporate rigid actions into $\mathcal{LP}(\mathbf{P})$ (which is constructed using non-rigid operations only) as follows.

(Step 1) Suppose $\mathrm{Pre}(\rho_i) = \{\mathtt{p_1}(\vec{\mathtt{t}}_1), \ldots, \mathtt{p_s}(\vec{\mathtt{t}}_s)\}$. Then add the rule

$$\mathtt{must\_fire}(\mathtt{J}, \mathtt{Name}(\rho_\mathtt{i})) \quad \leftarrow \quad \mathtt{p_1}(\vec{\mathtt{t}}_1)\,\&\,\ldots\,\&\,\mathtt{p_s}(\vec{\mathtt{t}}_s)$$

to $\mathcal{LP}(\mathbf{P})$. Do this for all rigid actions ρ_i, $1 \leq i \leq k$.

(Step 2) For each $1 \leq i \leq k$, if $\mathtt{p}(\vec{\mathtt{t}}) \in \mathrm{Add}(\rho_i)$, then add the clause

$$\mathtt{add}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}})) \quad \leftarrow \quad \mathtt{must\_fire}(\mathtt{J}, \mathtt{Name}(\rho_\mathtt{i}))$$

to $\mathcal{LP}(\mathbf{P})$. Similarly, if $\mathtt{p}(\vec{\mathtt{t}}) \in \mathrm{Del}(\rho_i)$, then add the clause

$$\mathtt{del}(\mathtt{J}, \mathtt{p}(\vec{\mathtt{t}})) \quad \leftarrow \quad \mathtt{must\_fire}(\mathtt{J}, \mathtt{Name}(\rho_\mathtt{i}))$$

to $\mathcal{LP}(\mathbf{P})$.

The choice program that results by incorporating these rigid actions as above (adding them to $\mathcal{CH}(\mathbf{P})$ instead of to $\mathcal{LP}(\mathbf{P})$ as above) can easily be seen to be locally stratified modulo choice, and hence, the above construction (obtained by adding the above-mentioned clauses to $\mathcal{LP}(\mathbf{P})$) is guaranteed to possess at least one stable model. In addition, it is easy to verify that if j is any integer and if all the pre-conditions of a rigid action, ρ_i, are satisfied at stage j in some stable model M, then $\mathtt{must\_fire}(j, \mathtt{Name}(\rho_i))$ must be true in M as well, and hence, its effects will manifest themselves in the stable model M as well due to the clauses we have added to $\mathcal{LP}(\mathbf{P})$ above.

An important point to note is that rigid actions are fired using the `must_fire` predicate, not the `fired` predicate. Hence, $\mathtt{fired}(\mathtt{Name}(\alpha)$ is never ever true of any rigid action as formulated above. The implication of this is that one or more rigid actions can occur as "background" actions, concurrently with non-rigid actions. In other words, at any given point t in time, the above formalization allows a set of rigid actions (possibly empty, possibly containing just one element, or possibly several) to execute concurrently with a single non-rigid action. As the purpose of rigid actions is to model situations where certain actions are "forced" to be executed when their pre-conditions are satisfied, concurrent execution of rigid actions is a must (and is satisfied by the above formalization).

6 An Alternative Formulation

The add-list and the delete-list of an atom are now represented by rules (Rules(2) and (3)); however, for the sake of implementational efficiency, the add and delete lists can be represented as binary relations of unit clauses. Executing queries to a flat set of binary relations is more efficient than executing queries to rules with implicationds. Instead of the predicate symbols add and del, one may now have predicate symbols add_1 and del_1. If α is an action, and $\mathtt{p}(\vec{\mathtt{t}})$ is in its add-list, then the unit clause

$$\mathtt{add\_1}(\mathtt{Name}(\alpha), \mathtt{p}(\vec{\mathtt{t}})) \quad \leftarrow$$

can be inserted in $\mathcal{CH}(\mathbf{P})$ instead of the clause (2). Similarly, if $\mathtt{q}(\vec{\mathtt{s}})$ is in α's delete list, then the clause

$$\mathtt{add\_1}(\mathtt{Name}(\alpha), \mathtt{q}(\vec{\mathtt{s}})) \quad \leftarrow$$

can be inserted in $\mathcal{CH}(\mathbf{P})$ instead of the clause (3).

In view of the representation of effects (adds and deletes) by binary relations, the axioms defining add and del need to be redefined using the following clauses.

$$\begin{array}{lcl} \mathtt{add}(\mathtt{J},\mathtt{C}) & \leftarrow & \mathtt{fired}(\mathtt{J},\mathtt{Name}(\alpha))\,\&\,\mathtt{add\_1}(\mathtt{Name}(\alpha,\mathtt{C})). \\ \mathtt{del}(\mathtt{J},\mathtt{C}) & \leftarrow & \mathtt{fired}(\mathtt{J},\mathtt{Name}(\alpha))\,\&\,\mathtt{del\_1}(\mathtt{Name}(\alpha,\mathtt{C})). \end{array}$$

It is easy to see that this alternative formulation preserves the same properties that $\mathcal{CH}(\mathbf{P})$ and $\mathcal{LP}(\mathbf{P})$ possessed.

7 Conclusions

In this paper, we have shown that there is a translation (that can be performed in linear-time) which, given any planning domain as input, will produce a logic program with negation as output such that a goal can be achieved from the planning domain iff that goal is true in a stable model of the logic program obtained by the specified translation.

In the real-world, what is specified in a planning domain as being the "initial" state can well be erroneous. Such errors may occur because of a multiplicity of reasons such as: the original state was just incorrectly represented, an external agent changed the state and this change was not detected till some time later, etc. We have shown that such "surprises" can be neatly captured in our framework as a database update.

Finally, most theoretical models of planning domains do not contain facilities whereby an action has to be executed whenever its pre-conditions are satisfied (though many implementations do have this facility). Typically, planning domains pick one such "firable" action. However, in certain domains, it is critically necessary that certain types of actions (which we term as "rigid" actions) be executed as soon as their pre-conditions become true. We have shown how our framework can be used to represent such actions.

Due to space limitations, we have not been able to include a detailed example illustrating the theory underlying this paper. The user interested in a more comprehensive description is referred to [12].

Acknowledgements. We are grateful to Antonio Brogi, Jim Hendler and Kutluhan Erol for comments on a previous version of this manuscript.

Acknowledgements

This work was supported by the Army Research Office under grant number DAAL-03-92-G-0225, by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, by an NSF Young Investigator award IRI-93-57756, by ARPA Order A 716/ Rome Laboratories Contract Number F30602-93-C-0241, and by Hughes Aircraft Corp./U.C. MICRO Program award 92-181.

References

[1] A. Bonner and M. Kifer. (1993) *Transaction Logic Programming*, Proc. 1993 Intl. Conf. on Logic Programming (ed. D.S. Warren), pps 257–279, MIT Press.

[2] D. Chimenti et. al. (1990) *The LDL System Prototype*, IEEE Trans. on Knowledge and Data Engineering, 2, 1, pps 76–90.

[3] P. M. Dung. (1993) *Representing Actions in Logic Programming and its Applications in Database Updates*, Proc. 1993 Intl. Conf. on Logic Programming (ed. D.S. Warren), pps 222–238, MIT Press.

[4] K. Erol, D.S. Nau and V.S. Subrahmanian. (1992) *On the Complexity of Domain-Independent Planning, Proc. AAAI-92*, MIT Press, July 1992.

[5] K. Erol, D.S. Nau and V.S. Subrahmanian. (1992) *Complexity, Decidability and Undecidability Results for Domain-Independent Planning.* accepted for publication in: Artificial Intelligence journal.

[6] M. Gelfond and V. Lifschitz. (1988) *Logic Programs with Classical Negation*, in: Proc. 7th International Conference on Logic Programming, eds. D.H.D. Warren and P. Szeredi, pps 579–597.

[7] M. Gelfond and V. Lifschitz. (1992) *Representing Actions in Extended Logic Programming*, Proc. 9th International Conference and Symposium on Logic Programming, ed. K. Apt.

[8] J. McCarthy and P. Hayes. (1969) *Some Philosophical Problems from the Standpoint of Artificial Intelligence*, in: Machine Intelligence 4 (eds. B. Meltzer and D. Michie), pps 463–502.

[9] J. Pinto and R. Reiter. (1993) *Temporal Reasoning in Logic Programming: A Case for the Situation Calculus*, Proc. 1993 Intl. Conf. on Logic Programming (ed. D.S. Warren), pps 203–221, MIT Press.

[10] T. Przymusinski. (1988) *On the Declarative and Procedural Semantics of Stratified Deductive Databases*, in J. Minker (ed.), "Foundations of Deductive Databases and Logic Programming," pps 193–216, Morgan-Kaufman.

[11] D. Sacca and C. Zaniolo. (1990) *Stable Models and Non-Determinism in Logic Programs with Negation*, Proc. 9th ACM Symp. on Principles of Database Systems.

[12] V.S. Subrahmanian and C. Zaniolo. (1994) *Database Updates and AI Planning Domains*, Tech. Report, Univ. of Maryland.

[13] C. Zaniolo. (1993) *A Unified Semantics for Active and Deductive Databases*, in "Rules in Database Systems," (N. Paton, ed.), Springer Verlag, 1994.